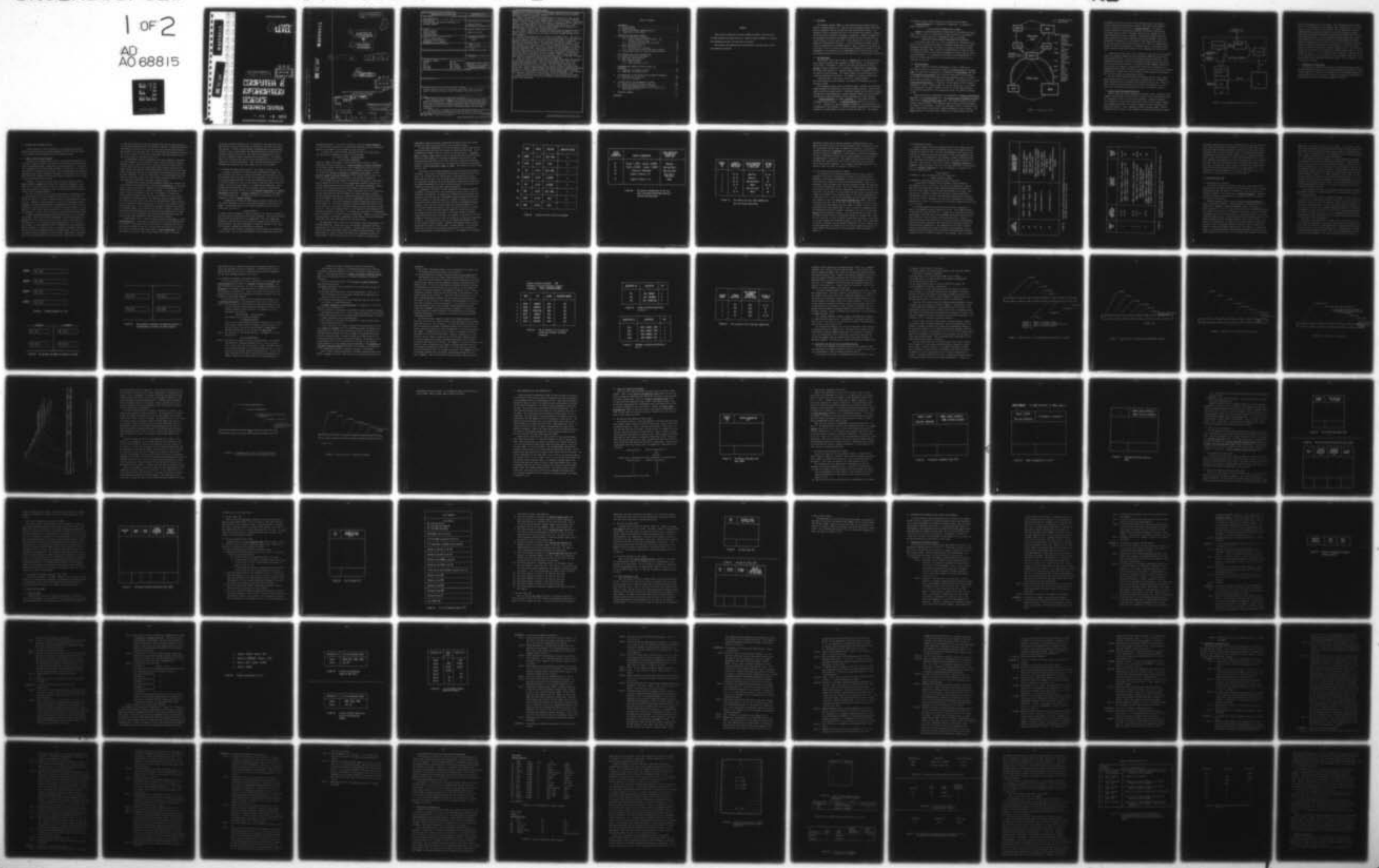
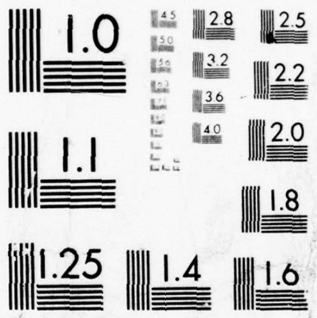
AD-A068 815 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 5/2
THE CLUSTERING AND SECURITY MECHANISMS OF A DATABASE COMPUTER (--ETC(U)
APR 79 J BANERJEE, D K HSIAO, J MENON N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-79-2 NL

1 OF 2
AD
A068815





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECHNICAL REPORT SERIES

LEVEL

# COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

14 OSU-CISRC-TR-79-2

1

6

# THE CLUSTERING AND SECURITY MECHANISMS OF A DATABASE COMPUTER (DBC)

BY

10 JAYANTA BANERJEE
DAVID K. HSIAO
JAISHANKAR MENON

9 Technical rept.,

12 118p.

15

Work Performed Under
Contract N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210

11 6 Apr[illegible]79

407 586 [illegible]

79 05 18 083

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>OSU-CISRC-TR-79-2 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>The Clustering and Security Mechanisms of a Database Computer | | 5. TYPE OF REPORT & PERIOD COVERED<br>Technical Report<br>6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Jayanta Banerjee<br>David K. Hsiao<br>Jaishankar Menon | | 8. CONTRACT OR GRANT NUMBER(s)<br>N00014-75-C-0573 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Office of Naval Research<br>Information Systems Program<br>Washington, D. C. 20360 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>4115-A1 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS | | 12. REPORT DATE<br>April 6, 1979<br>13. NUMBER OF PAGES<br>115 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

| | |
|---|---|
| Scientific Officer | DCC New York Area |
| ONR BRO | ONR 437 |
| ACO | ONR, Boston |
| NRL 2627 | ONR, Chicago |
| ONR 102IP | ONR, Pasadena |

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database computer, security, clustering, field-level security, security atoms, primary clustering attribute, secondary clustering attributes

10 to the 10th power

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The database computer (DBC) is a specialized back-end computer which is capable of managing data of $10^{10}$ bytes in size and supporting known data models such as relational, network, heirarchical and attribute-based models. It is also one of the first database machines to have a built-in security mechanism for access control and a clustering mechanism for performance enhancement.

In this report, we demonstrate how the important and essential functions of access control and clustering are carried out in DBC. Since these

next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) functions are carried out mostly in the database command and control processor (DBCCP) and the security filter processor (SFP), the report will be devoted largely to these two components of DBC.

DBCCP is responsible for regulating the entire operation of DBC, and for interfacing with the front-end computer systems. In the process of enforcing security and clustering records, DBCCP requires the use of a number of tables and lists. In order to speed up the table search, a subcomponent of SFP, namely the security and clustering unit (SCU), is presented with some details. SCU consists of a large number of parallel memory-processor pairs organized in a manner similar to that in the structure memory (SM).

The entire access control mechanism revolves around the important notion of security atoms. Briefly, this involves the protection of aggregates of data units, where the aggregates are definable by the file creator in terms of the data contents.

DBC can also protect data at the field (or attribute) level. The user of a file stored in DBC submits a list of fields that he wishes to access along with an access request. DBC can then check, from the tables that it maintains, whether the requested access on this combination of fields is allowed. If not, the request is rejected by DBC. Otherwise, DBC performs the required access. If the access is a 'retrieve' request, then the relevant records are retrieved from the mass memory (MM), which is the repository of the database, and sent to the post processor (PP), a subcomponent of SFP. PP then extracts, from these records, those fields that were requested by the user and sends them onto the front-end computer from which the user's request originated. The remainder of the fields in the retrieved records are discarded. Thus, access imprecision never results in the sense that no retrieved record has to be entirely discarded. Only parts of retrieved records may have to be masked out.

Since MM consists of moving-head disks and since the principal MM operation is searching, we would like to limit the search space in MM. This is done by using a clustering mechanism. The clustering mechanism of DBC also uses information provided by the file creator. The file creator must have some ideas of the types of queries that will be used to retrieve or delete records in his file. Based on this knowledge, he specifies a set of clustering attributes. Records are clustered together based on the values taken by these clustering attributes in the keywords of the records. With this mechanism, it is estimated that most user requests will require access to only one cylinder in MM.

Finally, the report goes into the details of the tables and lists needed to enforce security and to facilitate clustering, and the algorithms needed to support these mechanisms.

TABLE OF CONTENTS

1. BACKGROUND . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.1 DBC Data Model . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 DBC Architecture . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Control Flow During Command Execution . . . . . . . . . . . . 4
1.4 Organization of this Report . . . . . . . . . . . . . . . . . 6

2. SECURITY AND CLUSTERING CONCEPTS . . . . . . . . . . . . . . . . 7
2.1 Basis of DBC Security Mechanism . . . . . . . . . . . . . . . 7
2.1.1 Subfile and Record Level Security - The Concept of Security Atom . . . . . . . . . . . . . . . . 9
2.1.2 The Atom Access Privilege List (AAPL) . . . . . . . . . 15
2.1.3 Field-Level Security . . . . . . . . . . . . . . . . . . 16
2.2 Data Clustering in DBC . . . . . . . . . . . . . . . . . . . 19
2.2.1 Motivation for Clustering . . . . . . . . . . . . . . . 19
2.2.2 Clustering Descriptors and Logical Clusters . . . . . . 23
2.3 Introducing the Security and Clustering Mechanisms . . . . . 29

3. DATA ORGANIZATION IN THE STRUCTURE LOOP . . . . . . . . . . . . 40
3.1 Tables for Security Enforcement . . . . . . . . . . . . . . . 41
3.2 Tables for Data Clustering . . . . . . . . . . . . . . . . . 47
3.3 Other Table Structures . . . . . . . . . . . . . . . . . . . 49
3.3.1 Tables in DBCCP . . . . . . . . . . . . . . . . . . . . 49
3.3.2 Table Structures in IXU . . . . . . . . . . . . . . . . 55

4. ALGORITHMS FOR SUPPORTING ACCESS CONTROL AND CLUSTERING . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.1 Algorithms for Database Creation . . . . . . . . . . . . . . 58
4.2 Algorithms for Database Access . . . . . . . . . . . . . . . 75

5. AN ILLUSTRATION OF THE SECURITY AND CLUSTERING MECHANISMS . . . . 81
5.1 Table and List Construction . . . . . . . . . . . . . . . . . 81
5.2 Execution of Requests . . . . . . . . . . . . . . . . . . . . 90

6. A CLOSER LOOK AT THE ARCHITECTURAL ELEMENTS . . . . . . . . . . . 101
6.1 Certain Restrictions Placed on the User . . . . . . . . . . . 105
6.2 Adding and Changing Security Requirements . . . . . . . . . . 106
6.3 Some Miscellaneous Information in the Security and Clustering Unit (SCU) . . . . . . . . . . . . . . . . . . . . 109

7. CONCLUDING REMARKS . . . . . . . . . . . . . . . . . . . . . . . 110

REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111

## PREFACE

This work was supported by Contract N00014-75-C-0573, from the Office of Naval Research and supervised by Dr. David K. Hsiao, Professor of Computer and Information Science, The Ohio State University.

The contract was administrated and monitored by The Ohio State University Research Foundation.

## I. BACKGROUND

The database computer (DBC) is a specialized back-end computer which is capable of managing data of $10^{10}$ bytes in size and supporting known data models such as relational, network, hierarchical and attribute-based models. Any operation of DBC is concerned with one of the following four aspects - searching and retrieval, security, clustering, or updating. A number of papers [1-6] are available that motivate the design of DBC and discuss the search and retrieval aspects of DBC in some detail. The reader may wish to refer to these papers for clarification of details. A report discussing the update aspects of DBC is scheduled to be published shortly. What we intend to demonstrate in this report is how the security and clustering functions are carried out in DBC. Fundamental to our discussion is an understanding of the built-in data model and overall architecture of DBC which are dealt with in the following sub-sections.

### 1.1 DBC Data Model

The smallest unit of data in DBC is a keyword which is an attribute-value pair, where the attribute may represent the type, quality, or characteristic of the value. Information is stored in and retrieved from DBC in terms of records; a record is made up of a collection of keywords and a record body. The record body consists of a (possibly empty) string of characters which are not used for search purposes. For logical reasons, all the attributes in a record are required to be distinct. An example of a record is shown below:

(<Relation, EMP>, <Job, MGR>, <Dept, TOY>, <Salary, 30000>).

The record consists of four keywords. The value of the attribute Job, for instance, is MGR.

DBC recognizes several types of keywords: simple, security and clustering. Simple keywords are intended for search and retrieval purposes. Security keywords are used for access control. Clustering keywords are utilized for placing records with a high probability of being retrieved together in close proximity. Security and clustering keywords will be elaborated on in later sections.

A keyword predicate, or simply, predicate, is of the form (attribute, relational operator, value). A relational operator can be one of $\{=, \neq, >, \geq, <, \leq\}$. A keyword K is said to satisfy a predicate T if the attribute of K is identical to the attribute in T and the relation specified by the relational operator of T holds between the value of K and the value in T. For example, the keyword <Salary, 15000> satisfies the predicate (Salary>10000).

A query conjunction is a conjunction of predicates. An example of a query conjunction is:

(Salary>25000)∧(Job≠MGR)∧(Relation=EMP).

A query is a Boolean expression of predicates in the disjunctive normal form. Thus, a query is a disjunction of query conjunctions. An example of the types of queries that may be recognized by DBC is as follows:

((Dept=TOY)∧(Salary<10000))∨((Dept=BOOK)∧(Salary>50000)).

If the above query refers to a file of employees of a department store, then it will be satisfied by records of employees working either in the toy department and making less than 10,000, or working in the book department and making more than 50,000. Queries, as defined above, are used not only to search and retrieve records from the database, but also to specify protection requirements. It is important to note that by utilizing queries as a means for security specification, any retrievable data in DBC can also be protected for security.

## 1.2 DBC Architecture

Figure 1 shows the schematic architecture of DBC. It consists of two loops of memories and processors, namely the structure loop and the data loop. The data loop is composed of two components: the mass memory (MM), and the security filter processor (SFP). MM is the repository of the database and is made of modified moving-head disks where all the tracks of a cylinder may be read in parallel in a single disk revolution. This modification is termed tracks-in-parallel-readout. In addition, the mass memory of DBC is content-addressable. Given a cylinder number and a query conjunction, it is possible to content-search the entire cylinder "on the fly" in the same revolution. For a detailed description of the organization of MM, the reader is referred to [6].

The structure loop is composed of four components: the keyword transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP) and the index translation unit (IXU). KXU converts keywords into their internal representations. SM is primarily used to store, retrieve and update the indices of the database. Indices are maintained in SM as a directory. Each entry in the directory consists of a keyword followed by a set of indices. An index consists of a cylinder number (actually, the index contains some more

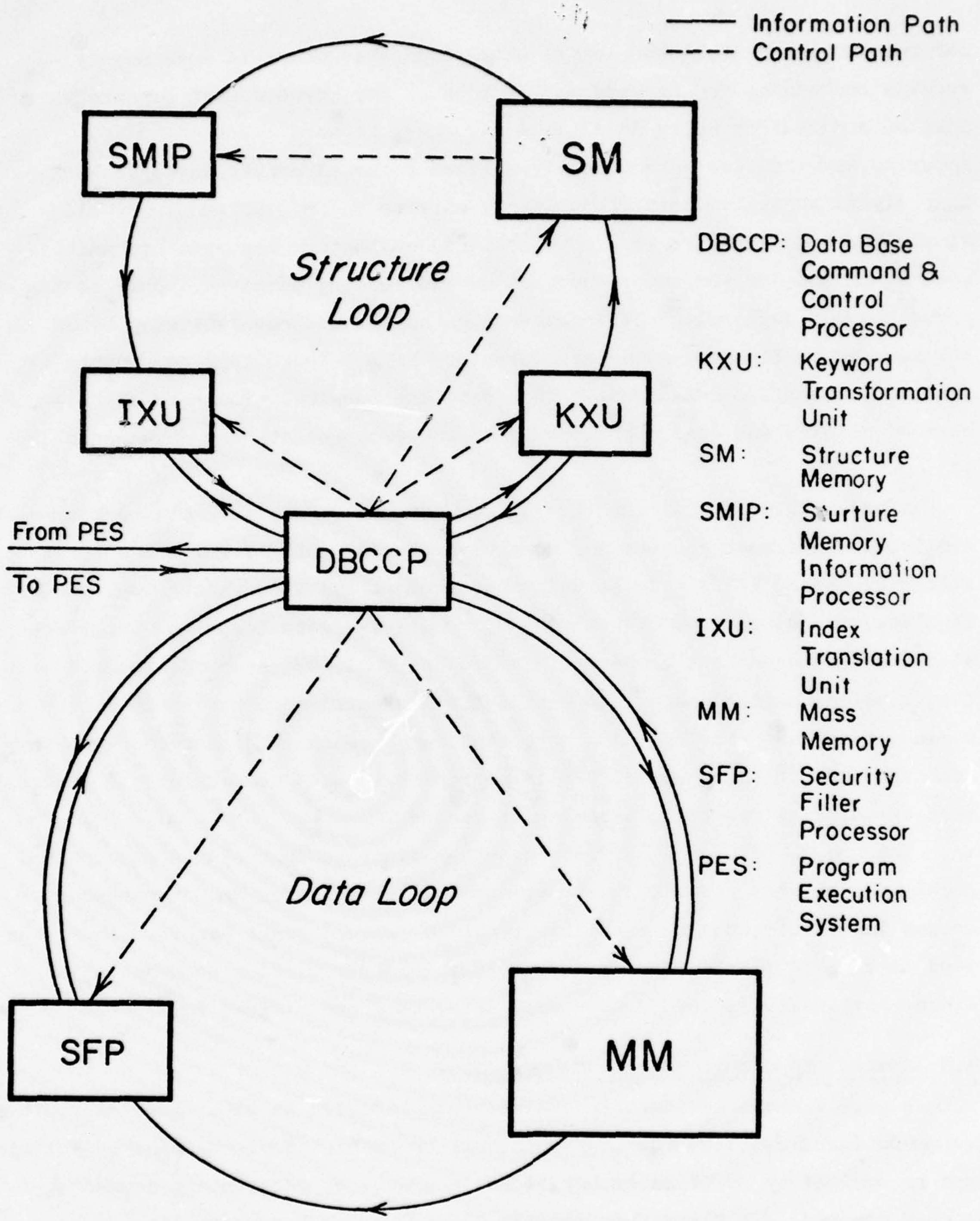

Information Path
Control Path
SMIP
SM
Structure Loop
IXU
KXU
From PES
To PES
DBCCP
Data Loop
SFP
MM
DBCCP: Data Base Command & Control Processor
KXU: Keyword Transformation Unit
SM: Structure Memory
SMIP: Sturture Memory Information Processor
IXU: Index Translation Unit
MM: Mass Memory
SFP: Security Filter Processor
PES: Program Execution System

FIGURE 1. Architecture of DBC

information as we shall see later) which indicates where in mass memory records containing the keyword may be found. Any keyword that appears as part of a directory entry in SM is a directory keyword. For example, security keywords are automatically defined to be directory keywords and they always appear as part of directory entries in SM. However, not all simple keywords are directory keywords. Non-directory keywords are mainly used by MM and SFP for record comparison and sorting purposes. SMIP is responsible for performing set intersections on the indices retrieved by SM. IXU is used to decode the indices output by SMIP. These four components are designed to operate concurrently in a pipeline fashion. The hardware organization, details and design philosophy of these components are documented in [5].

A major part of this report will concentrate on the role of two components - namely, the database command and control processor (DBCCP) and the security filter processor (SFP) - in security enforcement and data clustering. DBCCP regulates the operations of both the structure and data loops and interfaces with the front-end host computer. It processes all DBC commands received from the front-end host computer, schedules their execution on the basis of the command type and priority, and routes the response to the front-end host computer. Additionally, it makes use of SFP to search the tables needed for the enforcement of security and the clustering of records.

The use of DBCCP and SFP provides the advantage that all access control checks can be made prior to the retrieval of records from MM. Consequently, access imprecision due to redundant record retrieval never occurs. Thus, the need to discard retrieved records due to security violation does not arise since they are retrieved from MM only after they are cleared for access.

## 1.3 Control Flow During Command Execution

Figure 2 shows how certain commands are executed in DBC. Basically, these commands forwarded from the front-end host computer (in pre-determined formats), are recognized by DBCCP as either access commands or preparatory commands. Access commands are those that require DBC to access the mass memory; preparatory commands precede and follow access commands and convey important housekeeping information. Access commands go through a security check. Having undergone security checks, access commands are translated by DBCCP into orders

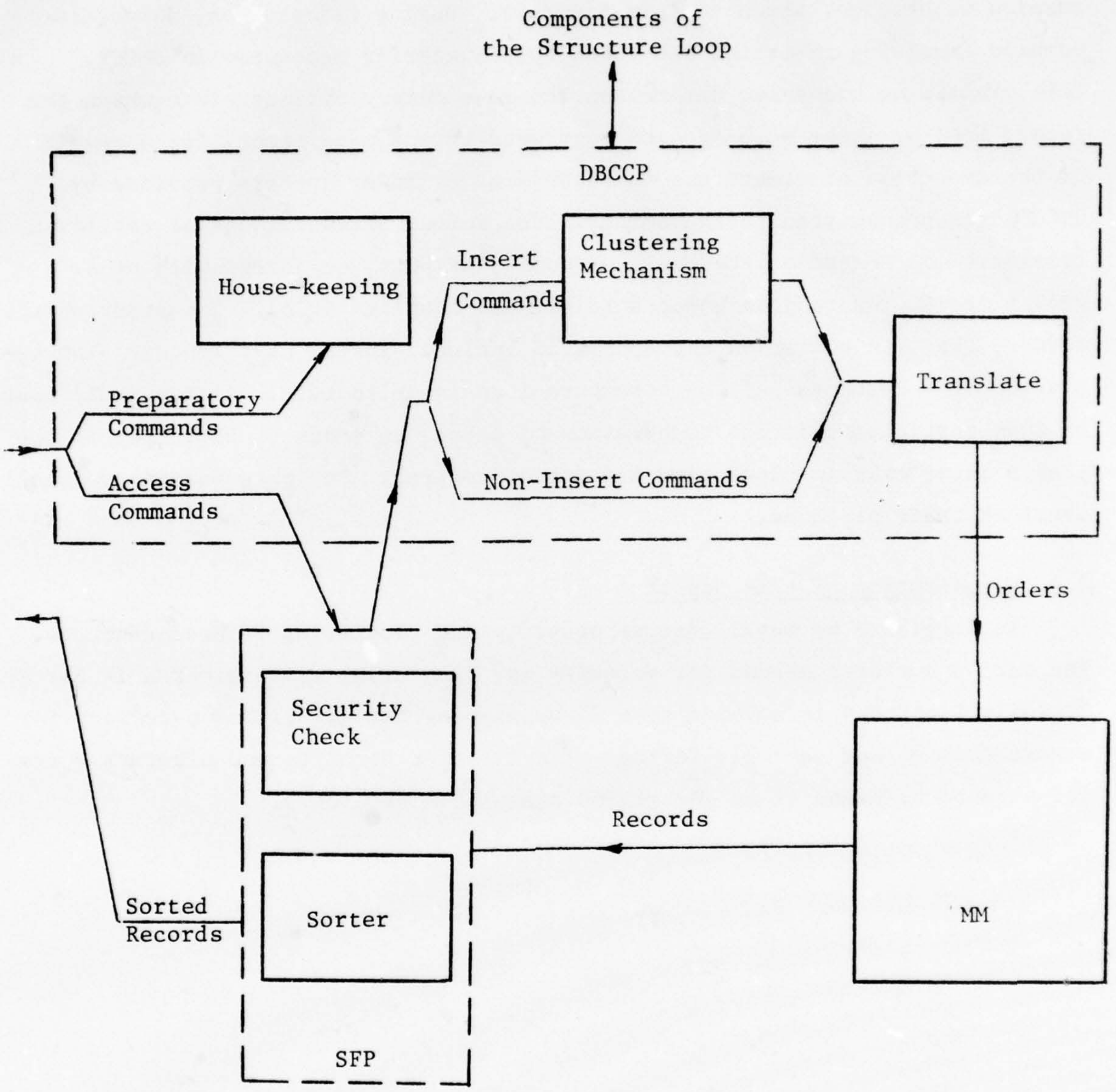


FIGURE 2. Access Command Execution in the Data Loop

that can be processed by the mass memory. This translation process requires the use of the components in the structure loop. Nevertheless, for our discussion we have not shown them in Figure 2. During translation, an access command involving insertion activates the clustering mechanism in DBCCP. This clustering mechanism determines the mass memory cylinder into which the record must be inserted. Records retrieved by the mass memory (as a result of the execution of orders and the supply of cylinder numbers provided by DBCCP) are transmitted to the sorter. The sorter allows groups of retrieved records to be sorted on the basis of some attribute, or joined with other groups of records (a join being a relational equality join). The study of the sorter, the join operation and update is included in the next report. The exclusion of components of the structure loop is deliberate. Although all four of them participate in the translation of access commands of DBC, they do not play a major role in clustering and access control. For this reason, we shall overlook their presence.

## 1.4 Organization of this Report

In Section 2 we shall discuss security and clustering related concepts. The data structures needed for security and clustering are presented in Section 3, while Section 4 is devoted to a discussion of the algorithms necessary for access control and data clustering. Finally, the concepts and algorithms are illustrated by means of an integrated example in Section 5.

## 2. SECURITY AND CLUSTERING CONCEPTS

In this section, we shall concentrate on the concepts that form the basis of the security and clustering mechanisms of DBC. This will lead us to the details of these mechanisms in the following sections.

### 2.1 Basis of DBC Security Mechanism

To protect the confidentiality of data, two principal problems must be resolved by computer systems. The first is to conceal the data in user-computer and computer-computer communications using encryption [7]. the second is to determine who can perform what operations on which data. The latter is the problem of providing access control. DBC is concerned with providing an efficient access control mechanism. Encryption, if necessary, may be supported by the use of encryption and decryption boxes at either end of the communication line between DBC and its host.

Naturally, it is up to the file creator to specify who can perform what operations on which data for his file. What we, as designers, would like to do is provide him with an easy, yet efficient, mechanism to convey these three pieces of information to DBC. We shall describe this mechanism of DBC in the following paragraphs.

The user can convey two of the above three pieces of information without any difficulty. It is easy enough for the file creator to be able to specify the names of the users to whom he wishes to give permission to access his file. Likewise, the specification of a list of the allowable accesses (examples of accesses being read-only, read-and-modify, etc.) for each user permitted to access the file does not provide any special problem. So, the problem of access control really boils down to the problem of how to specify the portions of a file to which accesses are allowed for the different users.

Different database management systems have different ways of tackling the problem. Many avoid the problem altogether by allowing an access on the entire file, or not allowing the access at all. This, as the reader will well appreciate, is not a powerful enough mechanism since it is lacking in two respects. We shall demonstrate the two weaknesses of this mechanism by means of examples. Consider the case of a database consisting of employee records, where all the employees work at the White House. Of course, one of the records contains information on the president, Jimmy Carter. Naturally, the file creator would not wish to allow anybody, other than the president, to read that particular record even though he would like to allow many users to read some of the other records in the file. However, the mechanism described above does not provide the file creator with a way to allow 'read' access on all records of the file except one. He is faced with the unpleasant task of having to deny a person 'read' access to the entire file, just because that person is not allowed to read <u>one</u> record in it.

We shall again utilize the database of the above example to illustrate the second weakness of this mechanism. Every record in the above database consists of, say, a Name field, a Job Position field and a Salary field. It is very likely that the file creator would want to allow a particular user 'read' access to the Name and Job Position fields, but not to the Salary field. This is because institutions are traditionally very sensitive about what they pay their employees. If the file creator were to use the mechanism described above, he would have to deny users 'read' access to entire records just because they are not allowed to read parts of these records.

A mechanism used by many database management systems that overcomes one of the weaknesses described above is called the <u>view mechanism</u>. In this mechanism, a user is not provided a complete 'picture' of the database. He is only provided with a window or view of the database. A view consists of a subset of all the attributes of the file. For example, to overcome the second weakness described above, the user is provided with a view that does not contain the attribute Salary. He, therefore, views an employee record as consisting of only a Name field and a Job Position field. Thus, he has no way of accessing the Salary field, since, as far as he is concerned, the field does not even exist. For individual users, separate views may be specified for different subsets of the attributes of the file. In DBC, we have a security specification mechanism which is simpler than the specification of views. Yet, it can be used to implement views with very good performance since the security enforcement mechanism of DBC is supported by hardware.

Another method that is sometimes used for security enforcement is called <u>query modification</u>. A user query is executed after it has been modified with a collection of previously-specified conjunctions of predicates that define the records accessible by the user. The way that the execution of the modified query is carried out causes some degree of <u>access imprecision</u>, i.e., many records are actually retrieved and examined as a result of the user query, but are later rejected (i.e., not passed on to the user) since they are found to violate the conjunctions of predicates that define the accessible records . In DBC, we shall provide a mechanism which first determines a priori a list of record groups (or compartments known as atoms) that are actually accessible by the user. Only these record groups, and no more, need to be examined; thereby eliminating access imprecision. The original query (i.e., the given user query) is now evaluated only on the records belonging to the above record groups. Thus, the time-consuming process of checking each individual record for access control is also avoided.

### 2.1.1 Subfile and Record Level Security - The Concept of Security Atom

We wish to incorporate in DBC a security-enforcement mechanism that will allow the file creator to protect his database at the sub-file, record and attribute levels but which will not be as unwieldy as the view mechanism or as time-consuming and costly as the query-modification scheme. To be able to refer to portions of a file, a file creator in DBC uses query conjunctions. A query conjunction, we may recall, is a conjunction of predicates. Query conjunctions and predicates used in this manner, solely for partitioning the database for access control, are called security conjunctions and security predicates respectively. A security attribute is an attribute that appears in a security predicate. A keyword of a record that satisfies a security predicate is called a security keyword.

To demonstrate how security conjunctions are used to partition a file into record aggregates, we use the same sample database of the previous examples. Let us assume that the file creator specifies only one security conjunction, and this is:

(Salary>15,000).

In order to keep the example as simple as possible, the security conjunction has been chosen to contain only one security predicate. This could, potentially, partition the database into two compartments. The first compartment contains all records, if any, which contains a keyword that satisfies the above predicate. The second compartment contains all records, if any, which do not contain a keyword that satisfies the predicate.

Along with each security conjunction, the file creator also specifies, for each user, a list of accesses permitted on all records that satisfy the security conjunction. This list of accesses is called an <u>access descriptor</u>. For each security conjunction, the file creator specifies p access descriptors, where p is the number of users allowed to access the file.

To take a more complex example, consider that the file creator specifies two security conjunctions S1 and S2 as shown below:

S1: (Salary$\geq$15,000) $\wedge$ (Salary$\leq$30,000).

S2: (Job Position=SECRETARY).

Before we complete the example, we shall define what we mean by a record satisfying a security conjunction. A <u>record satisfies a security conjunction</u> if the record contains keywords that satisfy <u>every</u> security predicate in the security conjunction. If a record satisfies a security conjunction, then we say that the <u>security conjunction is satisfied by the record</u>. A record <u>does not satisfy a security conjunction</u> if the record does not contain keywords to satisfy <u>some</u> security predicate in the security conjunction.

In the example being developed, four potential partitions or compartments may be created for the file, The first compartment contains records of secretaries earning between $15,000 and $30,000. The second compartment contains records of secretaries earning less than $15,000 or more than $30,000, and the third compartment contains records of employees who are not secretaries and who earn between $15,000 and $30,000, The fourth compartment contains records of employees who are not secretaries and who earn less than $15,000 or more than $30,000. More formally, the first compartment contains records which satisfy S1 and S2. The second compartment contains records that satisfy S2 but do not satisfy S1. The third compartment contains records that satisfy S1 but do not satisfy S2. The fourth compartment contains records that satisfy neither S1 nor S2. Thus, the compartment that a record belongs to is determined by the security conjunctions it satisfies. It follows that a maximum of $2^n$ compartments may be created for a file with n security conjunctions. However, in a real database, the total number of non-empty compartments is much smaller than this theoretical upper limit since many compartments do not have any records.

The compartment that a record of a file belongs to may be determined by the following algorithm. Assume that a set S of security conjunctions has been specified for the file by the file creator. Going through the set S, we determine for the record the largest subset of S that the record satisfies. This unique subset of security conjunctions defines the compartment to which the record belongs. Notice that DBC carries out the function of forming these compartments using the security conjunctions specified by the file creator, and the file creator and users need not even be aware of them.

The non-empty compartments of a file created as a result of the security conjunctions specified by the file creator are called security atoms. In the aforementioned example, there are at most four security atoms. A _security atom_, or simply an _atom_, defines a set of records such that all these records satisfy the same unique set of security conjunctions (and not any of the other security conjunctions of the file); this set of security conjunctions is said to define the security atom. As record sets, atoms have an important property - _no two atoms have a record in common_. The concept of security atoms is due to [8].

For any user, the accesses permitted on a security atom are determined as follows. The security conjunctions that define the atom are first identified. Each of these security conjunctions is associated with an access descriptor for each user. For all these security conjunctions, therefore, it is possible to identify the access descriptors relevant to the given user. The accesses permitted to the user on the given security atom are then determined by intersecting the above access descriptors.

In Figures 3a, 3b, and 3c we have illustrated the concept by means of an example. Figure 3a shows a file consisting of eight records, here each record has four keywords. In Figure 3b, we indicate the five security conjunctions of the file and the corresponding access descriptors specified by the file creator for one particular user. Finally, in Figure 3c, we show the security atoms created for the file by DBC and corresponding permitted accesses. Note, that only six atoms have been created.although up to $2^5$ (=32) security atoms are possible. This is because, typically, the number of atoms of a file is much less than the number of records in the file. The figure also shows the composition of the atoms in terms of security conjunctions and in terms of records. For example, we see that the record R1 belongs to the security atom 1, which is defined by the security conjunctions S1 and S4. To see why this is true, let us consider the keywords of R1. R1 has a keyword <Salary, 20,000>. Consider this keyword and the security conjunction S1, which is $(\text{Salary}\geq 1{,}000) \wedge (\text{Salary}\leq 20{,}000)$. Since this keyword of R1 satisfies all the security predicates in S1, R1 satisfies S1. R1 has another keyword <Length-Of-Service, 6>. This keyword of R1 obviously

|  | NAME | SALARY | POSITION | LENGTH-OF-SERVICE |
|---|---|---|---|---|
| R1. | MENON | 19,000 | ASST. PROF. | 6 |
| R2. | HSIAO | 15,000 | PROF. | 12 |
| R3. | JAY | 24,000 | ASST. PROF. | 6 |
| R4. | LORENZO | 13,000 | LECTURER | 6 |
| R5. | JACK | 16,000 | LECTURER | 9 |
| R6. | PAUL | 18,000 | ASST. PROF. | 9 |
| R7. | KERR | 44,000 | PROF. | 12 |
| R8. | MARK | 45,000 | PROF. | 9 |

FIGURE 3A. COMPLETE VIEW OF A FILE WITH 8 RECORDS

| ID OF SECURITY CONJUNCTION | SECURITY CONJUNCTION | ACCESS DESCRIPTOR ASSIGNED TO A SINGLE USER |
|---|---|---|
| S1 | (SALARY $\geq$ 1,000) $\wedge$ (SALARY $\leq$ 20,000) | READ-ONLY |
| S2 | (SALARY $\geq$ 20,000) $\wedge$ (SALARY $\leq$ 30,000) | READ-AND-MODIFY |
| S3 | (POSITION = PROFESSOR) | READ-AND-INSERT |
| S4 | (LENGTH-OF-SERVICE = 6) | READ, MODIFY AND DELETE |
| S5 | (LENGTH-OF-SERVICE = 9) | (NONE) |

FIGURE 3B: THE SECURITY CONJUNCTIONS OF THE FILE. ALSO, THE ACCESS DESCRIPTORS SPECIFIED FOR ONE PARTICULAR USER.

| SECURITY ATOM ID | SECURITY CONJUNCTIONS APPLICABLE | ACCESS DESCRIPTOR OF THE ATOM FOR A SINGLE USER | RECORDS IN THE ATOM |
|---|---|---|---|
| 1 | S1, S4 | READ-ONLY | R1, R4 |
| 2 | S1, S3 | READ-ONLY | R2 |
| 3 | S2, S4 | READ AND MODIFY | R3 |
| 4 | S1, S5 | (NONE) | R5, R6 |
| 5 | S3 | READ AND INSERT | R7 |
| 6 | S3, S5 | (NONE) | R8 |

FIGURE 3C. THE ATOMS OF THE FILE, THEIR COMPOSITION, AND THEIR ACCESS DESCRIPTORS.

satisfies S4 which has one equality predicate (Length-Of-Service = 6). Hence, R1 satisfies S1 and S4, and hence belongs to the atom defined by these two conjunctions, namely, Atom 1 as depicted in Figure 3c. We say that a security conjunction is applicable to an atom if the records of the atom satisfy the security conjunction. In this example, S1 and S4 are applicable to Atom 1. Applicable conjunctions are listed alongside of the atom they are applicable to in Figure 3c. This example does not show the field or attribute level security which will also be incorporated as an integral part of DBC security mechanism. For an example that demonstrates this capability of DBC, the reader is referred to Section 2.1.3.

### 2.1.2 The Atom Access Privelege List (AAPL)

The discussion of the previous subsection tells us that in order to determine if an access is permitted on a record, it is enough for DBC to determine if the access is permitted on the security atom containing the record since all records in an atom are protected similarly. Thus, it is necessary to first determine those security conjunctions which are applicable to the atom containing the record. As we have stated earlier, the file creator specifies, with every security conjunction, the permitted accesses for every user. The accesses of the applicable security conjunctions may then be 'anded' to determine if the requested access on the record is permitted for a particular user. A list of atom identifiers and the corresponding permitted accesses is called the Atomic Access Privilege List (AAPL). One such list exists for every user of the file. Since the number of security atoms is expected to be much smaller than the number of records, this list will be relatively short, thus enabling DBC to scan it rapidly to determine whether the requested access may be granted. Furthermore, the list is created long before the user initiates a request. This is because DBC can create a list of atoms in terms of security conjunctions during file creation time. Knowing for each atom the set of security conjunctions applicable to that atom, it can then 'compute' the permitted accesses on that atom. Thus, the process of security enforcement has been greatly simplified. We note, finally, that security enforcement is carried out by the database command and control processor (DBCCP) using the security filter processor (SFP) as and when it needs to.

### 2.1.3 Field-level Security

Earlier, we have stated that along with each security conjunction, the file creator specifies p access descriptors where p is the number of users permitted to access the file. That is, corresponding to each security conjunction, he specifies one access descriptor for each user permitted to access the file. This mechanism is now extended to support field-level security. Along with each security conjunction, the file creator specifies, for each user, a set of pairs. Each pair is called a <u>field-level security descriptor</u>. These pairs are of the form (attribute combination, access descriptor). An <u>attribute combination</u> is a set of attributes. An example of an attribute combination is:

[Name, Salary].

An example of a pair specified by the file creator is:

([Name, Salary], No-Read).

An example of a security conjunction and a set of pairs is:

(Salary>100)----([Name, Salary], No-Read), ([Job], No-Modify).

The meaning of this is as follows. The user is not permitted to read the Name and Salary fields from records satisfying (Salary>100) although the the rest of the fields of these records are accessible to the user. Also, the user is not allowed to modify the Job field of records satisfying (Salary>100). Notice, that the access descriptors always specify accesses that are not allowed. Naturally, if an access is disallowed on an attribute combination $AC_i$, then it is disallowed on all attribute combinations which have $AC_i$ as a subset.

The mechanism which includes field-level security is explained by means of an example developed in Figures 3d and 3e. This example pertains to the same file shown in Figure 3a. In Figure 3d, we indicate the security conjunctions of the file and the corresponding field-level security descriptors specified by the file creator for one particular user. Figure 3e shows the security atoms created for the file by DBC and the corresponding field-level security descriptors which indicate the disallowed accesses on various attribute combinations. The composition of the atoms in terms of security conjunctions and in terms of records is also indicated. Let us explain how the field-level security descriptors indicating the disallowed accesses on Atom 1, as shown in Figure 3e, was obtained. Atom 1 is defined

| ID OF SECURITY CONJUNCTIONS | SECURITY CONJUNCTION | FIELD-LEVEL SECURITY DESCRIPTOR (FOR ONE PARTICULAR USER). |
|---|---|---|
| S1 | (SALARY $\geq$ 1,000) $\wedge$ (SALARY $\leq$ 20,000) | ([NAME, SALARY], NO-MODIFY), ([POSITION], NO-READ) |
| S2 | (SALARY $\geq$ 20,000) $\wedge$ (SALARY $\leq$ 30,000) | ([NAME, SALARY], NO-READ), ([POSITION], NO-READ) |
| S3 | (POSITION = PROFESSOR) | ([NAME, SALARY, POSITION], NO-READ), ([LENGTH-OF-SERVICE], NO-MODIFY) |
| S4 | (LENGTH-OF-SERVICE = 6) | ([NAME, SALARY], NO-READ), ([POSITION, LENGTH-OF-SERVICE], NO-MODIFY) |
| S5 | (LENGTH-OF-SERVICE = 9) | ([NAME], NO-MODIFY), ([SALARY, POSITION, LENGTH-OF-SERVICE], NO-READ) |

FIGURE 3D. THE SECURITY CONJUNCTIONS AND CORRESPONDING FIELD-LEVEL SECURITY DESCRIPTORS FOR A PARTICULAR USER FOR THE FILE OF FIGURE 3A.

| SECURITY ATOM ID | SECURITY CONJUNCTIONS APPLICABLE | FIELD-LEVEL SECURITY DESCRIPTOR | RECORDS IN THE ATOM |
|---|---|---|---|
| 1 | S1, S4 | ([NAME, SALARY], NO-READ), ([POSITION], NO-READ) | R1, R4 |
| 2 | S1, S3 | ([NAME, SALARY], NO-MODIFY), ([POSITION], NO-READ), ([LENGTH-OF-SERVICE], NO-MODIFY) | R2 |
| 3 | S2, S4 | ([NAME, SALARY], NO-READ), ([POSITION], NO-READ) | R3 |
| 4 | S1, S5 | ([NAME], NO-MODIFY), ([POSITION], NO-READ) | R5, R6 |
| 5 | S3 | ([NAME, SALARY, POSITION], NO-READ), ([LENGTH-OF-SERVICE], NO-MODIFY) | R7 |
| 6 | S3, S5 | ([NAME], NO-MODIFY), ([NAME, SALARY, POSITION], NO-READ), ([LENGTH-OF-SERVICE], NO-MODIFY), ([SALARY, POSITION, LENGTH-OF-SERVICE], NO-READ) | R8 |

FIGURE 3E. THE ATOMS OF A FILE, THEIR COMPOSITION, AND THE CORRESPONDING FIELD-LEVEL SECURITY DESCRIPTORS FOR A PARTICULAR USER.

by security conjunctions S1 and S4. From Figure 3d, we see that the field-level security descriptors corresponding to S1 do not allow 'read' access on both the Name and Salary attributes. Naturally, if the 'read' access is disallowed, then, so is the 'modify' access. The field-level descriptors corresponding to S4 allow the 'read' access but do not allow the 'modify' access on the same two attributes. Hence, the intersected set of descriptors which indicates the disallowed accesses on records in Atom 1, will disallow the 'read' (and hence the 'modify') access on the Name and Salary fields. Similarly, even though the 'read' access is allowed on the Position attribute by the field-level security descriptors corresponding to S4, it is disallowed by those corresponding to S1. Hence, the 'read' access is disallowed on the Position attribute of records in Atom 1.

## 2.2 Data Clustering in DBC

### 2.2.1 Motivation for Clustering

The need for placing data elements, which have a high probability of being retrieved and updated together, in close proximity in order to enhance performance has been the subject of research [14, 16]. Strategies for the 'optimal' placement of data are known as clustering strategies. Naturally, clustering strategies are dependent on the nature and use of the database for which they are intended.

Database management systems usually provide clustering mechanisms to support clustering strategies. These mechanisms are dependent on the technology used to implement a database management system. For example, in a RAP-like system [12, 13], the search and retrieval time is essentially constant with respect to the size of the database as long as the size does not exceed the memory capacity of the search processor. Therefore. there is no need for a RAP-like system to have a clustering mechanism if the database can be accommodated completely within the memory capacity of the search processor. If, however, the database size is larger than the memory capacity of the search processor, than a clustering mechanism is needed in order to minimize the number of loading and unloading operations per user request. As another example, in conventional software-laden database management systems the information brought from a disk device into high-speed main memory is contained in pages (e.g., a page is a fraction of a disk track). In such cases, a software clustering package is used at page creation time in order to ensure that these pages contain a high percentage of data relevant to a subsequent request. This will enable the sys [illegible]m to minimize the I/O traffic and buffering due to paging, thereby improving system performance.

In our design of DBC, the database is stored on modified moving-head disk devices with tracks-in-parallel-readout capability. In such devices, the search and retrieval time for data elements within a cylinder is a constant. This constant is the rotation time of the disk device. However, the arm movement time, which in most instances is substantially greater than the rotation time, varies roughly in proportion to the distance over which the access mechanism is moved. It is clear, therefore, that in order to derive the best possible performance from a modified moving-head device, it is important that we place the data elements in a way that minimizes the number of cylinders to be searched per user request. Here, we describe the two-level clustering mechanism of DBC which is to be used to support the clustering strategies devised by the users.

Let us motivate the concept of clustering and the resulting performance improvement by means of a simple example [4] of a file with four records as depicted in Figure 4a.

In Figure 4b we have shown an arbitrary placement of records in the two cylinders that have been made available to the file F. Now, if a query for retrieval is received in the form, "Retrieve records which satisfy the conjunction ($K1 \wedge K3$)", then DBC has to make two cylinder accesses. However, if the records are placed in the cylinders grouped according to the occurrence of keywords (i.e., K1, K2 and K3) in a record, then the resulting configuration will be as shown in Figure 4c. Such a configuration will facilitate the retrieval of all records which satisfy the given query conjunction with a single access to the mass memory.

The above diccussion implies two things: First, the creator of the file should have some ideas of the type of queries that will be made on the file, if he is interested in taking advantage of possible performance gain. Second, the system (i.e., DBC) provides him with a mechanism for effectively conveying that knowledge to DBC. While we, as system designers, cannot predict how much knowledge a file creator may have of his file usage, we

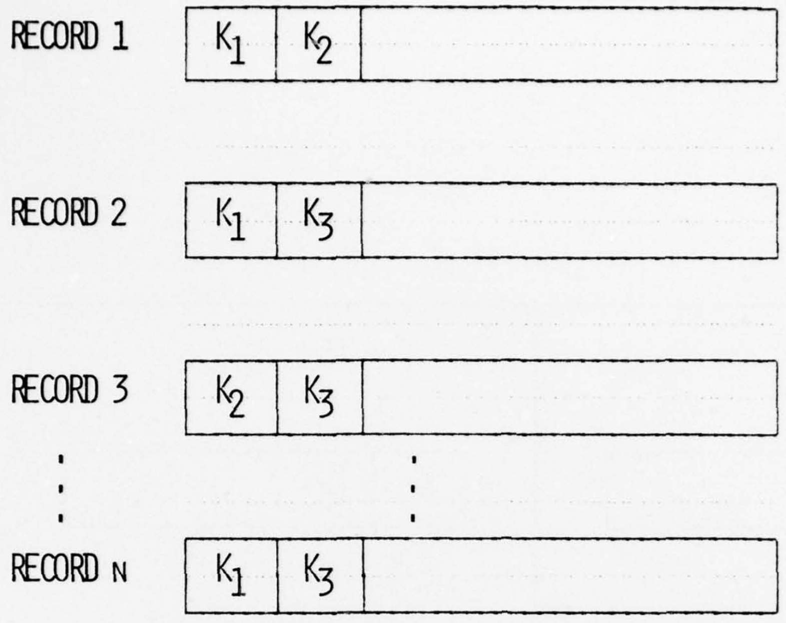


FIGURE 4A. RECORDS BELONGING TO A FILE

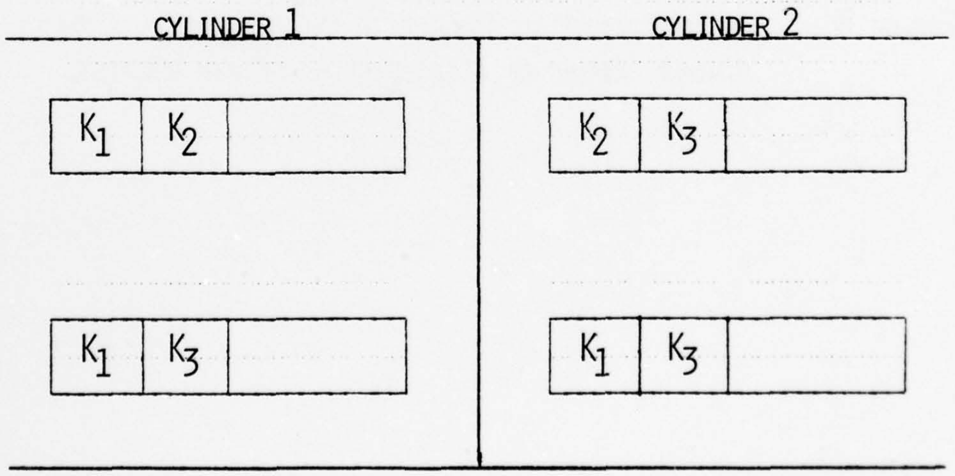


FIGURE 4B: AN ARBITRARY ASSIGNMENT OF RECORDS TO CYLINDER.

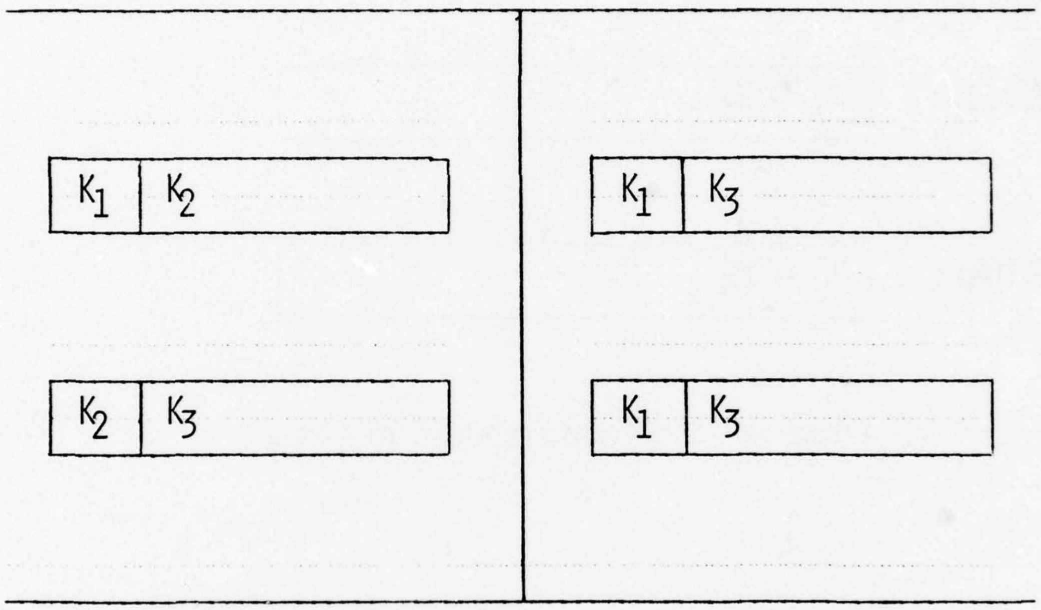


FIGURE 4c: AN ASSIGNMENT OF RECORDS TO CYLINDER WHICH RESULTS IN MINIMUM NUMBER OF ACCESSES FOR CERTAIN QUERIES.

must ensure that he is provided with an easy yet powerful mechanism to utilize that knowledge to his best advantage. The mechanism that we have adopted and shall describe here is capable of being naturally integrated into the query language available to the users.

### 2.2.2 Clustering Descriptors and Logical Clusters

A file is associated (by the file creator) with a single <u>primary clustering attribute</u> and any number of <u>secondary clustering attributes</u>. The latter attributes are specified as a list in an <u>order of importance</u>. The <u>importance</u> of a secondary clustering attribute is defined to be its relative position in the above list. Thus, we can talk of one secondary clustering attribute as being more important than another secondary clustering attribute for clustering purposes.

At the time of file creation, the file creator also specifies a set of <u>clustering descriptors</u>. These descriptors may be one of three types:

Type A: The descriptor is a conjunction of a less-than-or-equal-to predicate and a greater-than-or-equal-to predicate, such that the same attribute appears in both predicates. An example of a type-A descriptor is as follows:

$$((\text{Salary}\geq 2{,}000) \wedge (\text{Salary}\leq 10{,}000)).$$

More simply, this is written as follows:

$$(2{,}000\leq \text{Salary}\leq 10{,}000).$$

Thus, the file creator merely specifies an attribute (i.e., Salary) and a range of values ($2,000 - $10,000) for that attribute. We term the value to the left of the attribute the <u>lower limit</u> and the value to the right of the attribute the <u>upper limit</u>.

Type B: The descriptor is an equality predicate. An example of a type-B descriptor is:

(Position=PROFESSOR).

Type C: The descriptor consists of only an attribute name. Let us assume that there are n different keywords K1, K2, ..., Kn, in the records of this file, with this attribute. Then, this type-C descriptor is really equivalent to n type-B descriptors B1, B2, ..., Bn, where Bi is the equality predicate satisfied by Ki. In fact, this type-C descriptor will cause n different type-B descriptors to be formed. From now on, we shall refer to the type-B descriptors formed from a type-C descriptor as type-C sub-descriptors.

The attribute that appears in a clustering descriptor must either be the primary clustering attribute or one of the secondary clustering attributes. A clustering descriptor is a _primary (secondary) clustering descriptor_ if the corresponding attribute is a primary (secondary) clustering attribute.

The file creator must observe certain _rules in forming descriptors_. These are specified below:

(1) Ranges specified in type-A descriptors for a given attribute must be mutually exclusive.

(2) For every type-B descriptor of the form (attribute-1 = value-1), no type-A descriptor can have the same attribute and a range that contains its value (i.e., value-1).

(3) An attribute that appears in a type-C descriptor must not also appear in a type-A or a type-B descriptor.

A _primary (secondary) clustering keyword_ is a keyword of a record such that one of the following holds:

(a) The attribute of the keyword is specified in a type-A primary (secondary) clustering descriptor and the value is within the range of the descriptor.

(b) The attribute and value of the keyword match those specified in a type-B primary (secondary) clustering descriptor.

(c) The attribute of the keyword is specified in a type-C primary (secondary) clustering descriptor.

In all these cases, the primary (secondary) clustering keyword is said to be _derived_ or _derivable_ from the corresponding primary (secondary) clustering descriptor. Each primary clustering descriptor is associated with a maximum space requirement (in terms of number of cylinders) which indicates the estimated amount of storage required in the mass memory for all records having keywords derived from this descriptor. The estimate is only an approximate one, but better performance is obtained if the estimate closely reflects the actual maximum storage requirement. The _importance of a secondary clustering keyword_ is defined to be the same as the importance of the corresponding secondary clustering attribute of the keyword.

Every (primary clustering descriptor, secondary clustering descriptor) defines a _cluster_ of records. Each record in this cluster must satisfy two conditions.

(1) The primary clustering keyword of the record should be derivable from the primary clustering descriptor of the cluster.

(2) The most important secondary clustering keyword <u>of the record</u> should be derivable from the secondary clustering descriptor of the cluster.

Note that the most important secondary clustering keyword of one record need not be the same as the most important secondary clustering keyword of another record. In fact, the most important secondary clustering attribute in the keywords of one record need not be the same as the most important secondary clustering attribute in the keywords of another record. This is because the secondary clustering attributes present in the keywords of different records may be different. A record for insertion must have a primary clustering keyword; otherwise, the record will be rejected by DBC. The record is also rejected if its primary clustering keyword is not derivable from any primary clustering descriptor. If a record has no secondary clustering keyword, then a null secondary clustering keyword derivable from a null secondary clustering descriptor is assumed. If the most important secondary clustering keyword is not derivable from any secondary clustering descriptor, it is assumed to be derived from a null secondary clustering descriptor.

We illustrate these concepts by means of an example developed in Figures 5a, 5b, 5c and 5d. Figure 5a shows a file consisting of six records, where each record has four attributes. As can be seen from the figure, the user has specified the primary clustering attribute to be Job and the secondary clustering attributes in order of importance to be Salary and Department Number. In Figures 5b and 5c, the primary and secondary clustering descriptors of the file have been shown. Finally, in Figure 5d, we illustrate the various clusters formed for the file and their composition both in terms of primary and secondary clustering descriptors, and in terms of the records that make up the clusters. For example, the record R1 belongs to the cluster 1, which is defined by the primary clustering descriptor PY1 and the secondary clustering descriptor SEC1. To see why this is so, let us look at the keywords of R1. R1 contains the keyword <Job, MANAGER> and this is obviously derivable from the primary clustering descriptor PY1, i.e., (Job = MANAGER). Also, R1 contains the keyword <Salary, 1500> and this is

PRIMARY CLUSTERING ATTRIBUTE: JOB
SECONDARY CLUSTERING ATTRIBUTES IN ORDER OF
IMPORTANCE: SALARY, DAPARTMENT-NUMBER

| | NAME | JOB | SALARY | DEPARTMENT-NUMBER |
|---|---|---|---|---|
| 1. | HAYES | MANAGER | 1500 | 100 |
| 2. | NAYAK | ENGINEER | 2000 | 100 |
| 3. | BOONE | TECHNICIAN | 4000 | 200 |
| 4. | WHITE | MANAGER | 1200 | 700 |
| 5. | KLINE | ENGINEER | 2500 | 200 |
| 6. | PRICE | ENGINEER | 2500 | 200 |

FIGURE 5A. THE SIX RECORDS OF A FILE, AND ITS PRIMARY AND SECONDARY CLUSTERING ATTRIBUTES.

| DESCRIPTOR ID. | DESCRIPTOR | TYPE |
|---|---|---|
| PY1. | JOB = MANAGER | B |
| PY2. | JOB = ENGINEER | B |
| PY3. | JOB = TECHNICIAN | B |

FIGURE 5B. PRIMARY CLUSTERING DESCRIPTORS OF FILE F.

| DESCRIPTOR ID. | DESCRIPTOR | TYPE |
|---|---|---|
| SEC1. | $1000 \le$ SALARY $\le 2000$ | A |
| SEC2. | $2001 \le$ SALARY $\le 3000$ | A |
| SEC3. | DEPT. NUMBER = 100 | B |
| SEC4. | DEPT. NUMBER = 200 | B |

FIGURE 5C. SECONDARY CLUSTERING DESCRIPTORS OF FILE F.

| CLUSTER NUMBER | PRIMARY DESCRIPTOR | MOST IMPORTANT SECONDARY DESCRIPTOR FOR THE RECORD | RECORDS IN CLUSTER |
|---|---|---|---|
| 1 | PY1 | SEC1 | R1, R4 |
| 2 | PY2 | SEC1 | R2 |
| 3 | PY3 | SEC4 | R3 |
| 4 | PY2 | SEC2 | R5, R6 |

FIGURE 5D. THE CLUSTERS OF FILE F AND THEIR COMPOSITION.

derivable from the secondary clustering descriptor, SEC1, (i.e., $1000 \le \text{Salary} \le 2000$) since, 1500 lies between 1000 and 2000. It also contains the keyword <Department Number, 100>, which is derivable from another secondary clustering descriptor, SEC3. However, for clustering purposes we only consider the primary clustering descriptor PY1 and the secondary clustering descriptor SEC1 of R1 from which the primary clustering keyword of the record (i.e., <Job, MANAGER>) and the most important secondary clustering keyword of the record (i.e., <Salary, 1500>) are derivable. Hence, R1 belongs to the cluster defined by PY1 and SEC1.

It has been mentioned earlier that DBC can support three kinds of database organizations - relational, network, and hierarchical. We suggest that the primary clustering attribute for a relational database be 'relation'. The reason for clustering by relation is because database queries on relational databases will always involve one or more relations. In this manner, we will always be ensured that DBC will satisfy any given query involving a single relation by accessing, at most, a number of cylinders that is no greater than the number of cylinders required to store all the records in the given relation. Further clustering may be done based on the information on images and links [1, 9].

For a network (CODASYL) database, primary clustering may be based on the record location modes [2,10]. Alternatively, the primary clustering may be based on the record type [2,10] and the secondary clustering may be based on the record location modes.

For hierarchical databases such as the IMS databases, three different clustering policies are suggested [3]. The first policy clusters all DBC records which represent segments belonging to the same IMS root segment [11]. The second policy first clusters DBC records which represent all the IMS root segments and then clusters DBC records which represent all dependent segments. The third policy clusters the records by segment type.

## 2.3 Introducing the Security and Clustering Mechanisms

We shall now informally sketch the process of file creation in DBC (the algorithms are formally presented in Section 4), which will also clarify the mechanisms of data clustering and access control.

The creator first issues an, Open-database-file-for-creation, request.

All requests to DBC consist of six parts:

(1) FEC-ID, which is the identification number of the front-end computer from which the request originated,

(2) R-ID, which is the identification number of the request,

(3) R-CODE, which is a coded version of the request or command name,

(4) the priority of the request,

(5) the identification number of the user making the request, and

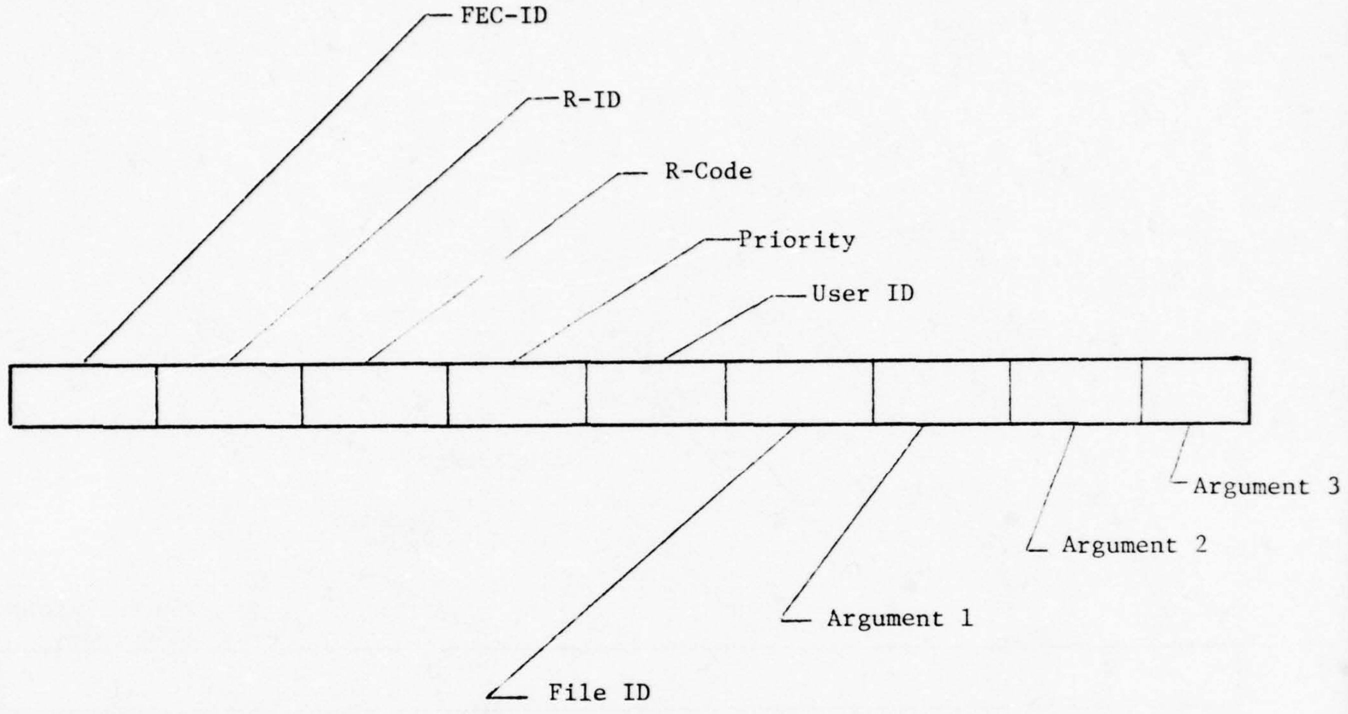
(6) the argument set (queries, records etc.)

A logical view of the 'Open-database-file-for-creation' request is shown in Figure 6. It provides information on the number of attributes the file is to have, the number of mass memory cylinders that need to be allocated for the file initially, and the number of cylinders that may be allocated additionally if the initial allocation is insufficient.

Next, the file creator issues the 'Load-attribute-information' request. A logical view of the request is shown in Figure 7. This request is used to provide DBC with information on the attributes used in the file. This information includes, for every attribute of the file, the maximum and minimum values the attribute may take, whether the attribute is a primary clustering or secondary clustering attribute, and the relative importance of the attribute if it is a secondary clustering attribute.

At this stage, the file creator issues a 'Load-descriptors' request, the format of which is shown in Figure 8a. Each of the descriptors in the request, as Figure 8b illustrates, includes the identification number of the descriptor, whether it is a primary clustering or secondary clustering descriptor, the type of the descriptor (i.e., A, B or C), the identification number of the attribute corresponding to the descriptor, and the range of values this attribute takes. Also, the maximum space requirement for primary clustering descriptors is indicated.

The file creator now loads all the necessary access control information by means of a 'Load-access-control-information' request, a logical view of which is shown in Figure 9a. This request identifies the users who are allowed to access the file, and assigns to each user U, K access privileges Ai ($1 \leq i \leq K$) corresponding to the K security conjunctions Si ($1 \leq i \leq K$). An access privilege Ai, for user U, specifies the kinds of accesses the user U may not make on all records satisfying Si. A view of an



| | |
|---|---|
| Argument 1: | Number of Attributes needed. |
| Argument 2: | Number of cylinders requested initially. |
| Argument 3: | Addition cylinders required. |

R-Code = $01_8$

FIGURE 6. Logical View of the 'Open-Database-File-for-Creation' Request

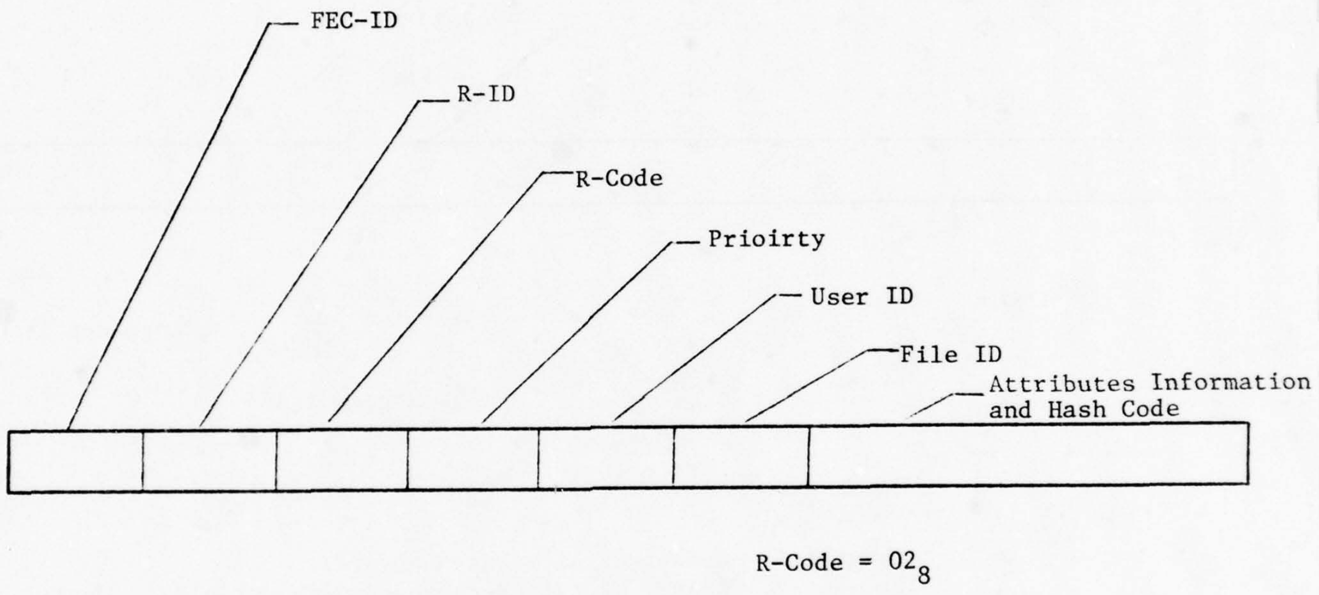


FIGURE 7. Logical View of a 'Load-Attribute-Information' Request

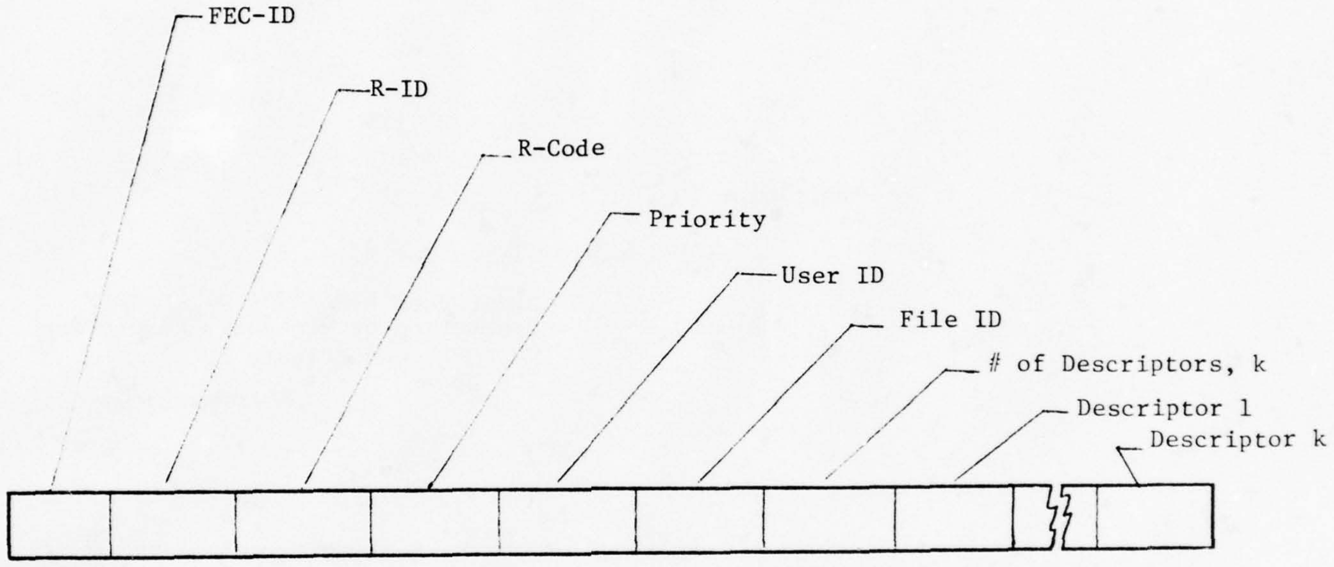


FIGURE 8a. Logical View of a 'Load-Descriptors' Request

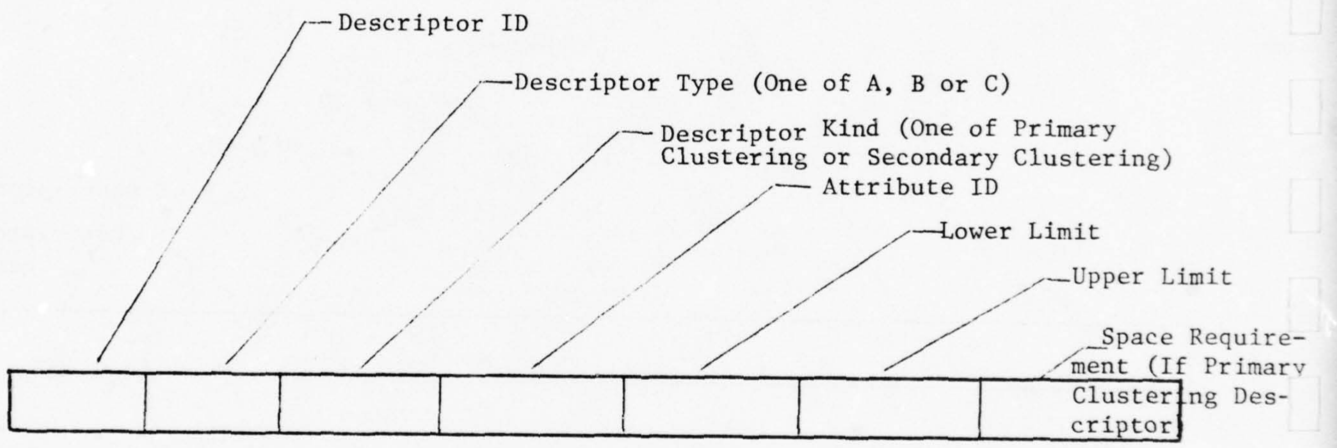
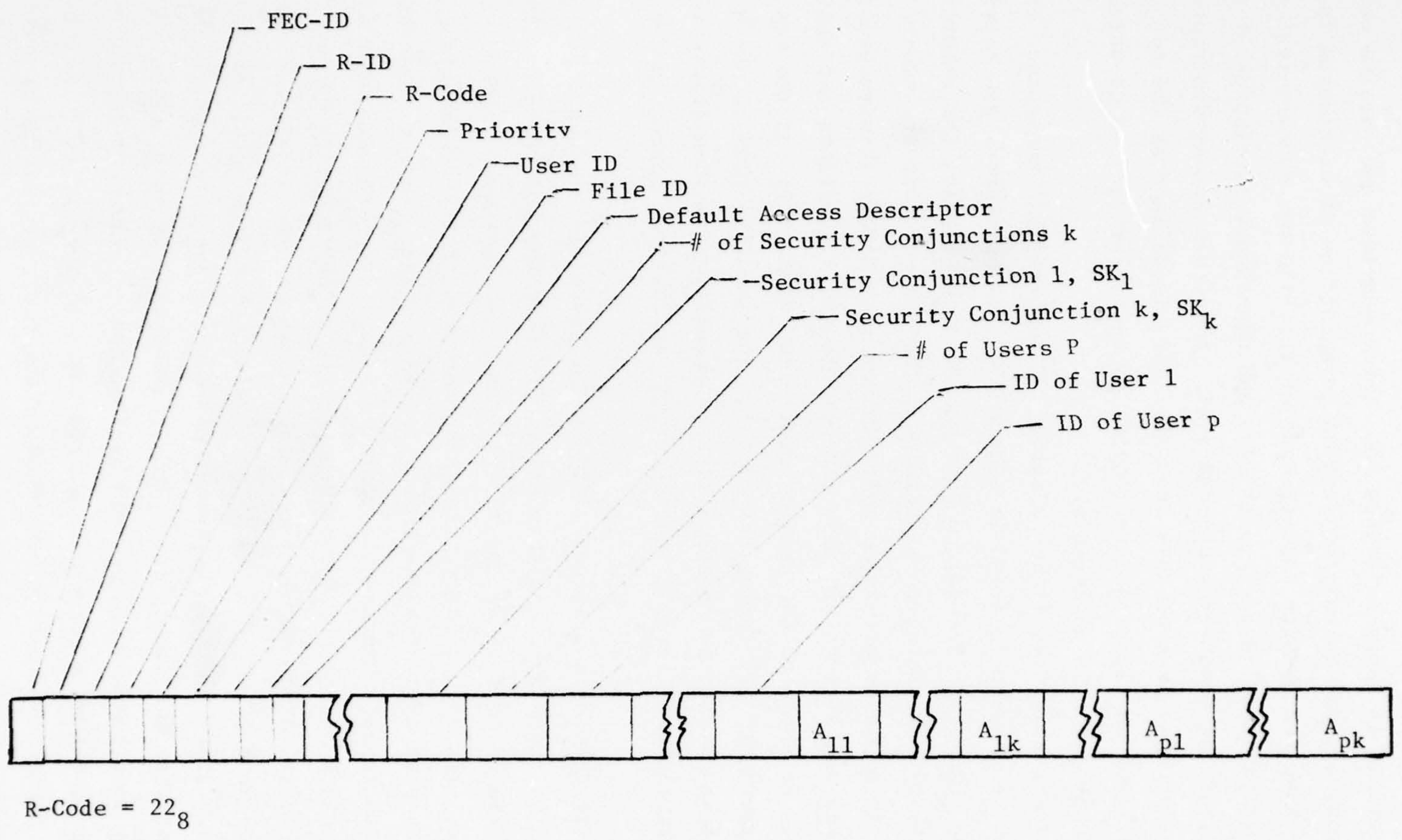

Descriptor ID
Descriptor Type (One of A, B or C)
Descriptor Kind (One of Primary Clustering or Secondary Clustering)
Attribute ID
Lower Limit
Upper Limit
Space Requirement (If Primary Clustering Descriptor)

FIGURE 8b. Logical View of a Descriptor



Aij: Access privileges allowed for user i on all records satisfying conjunction j.

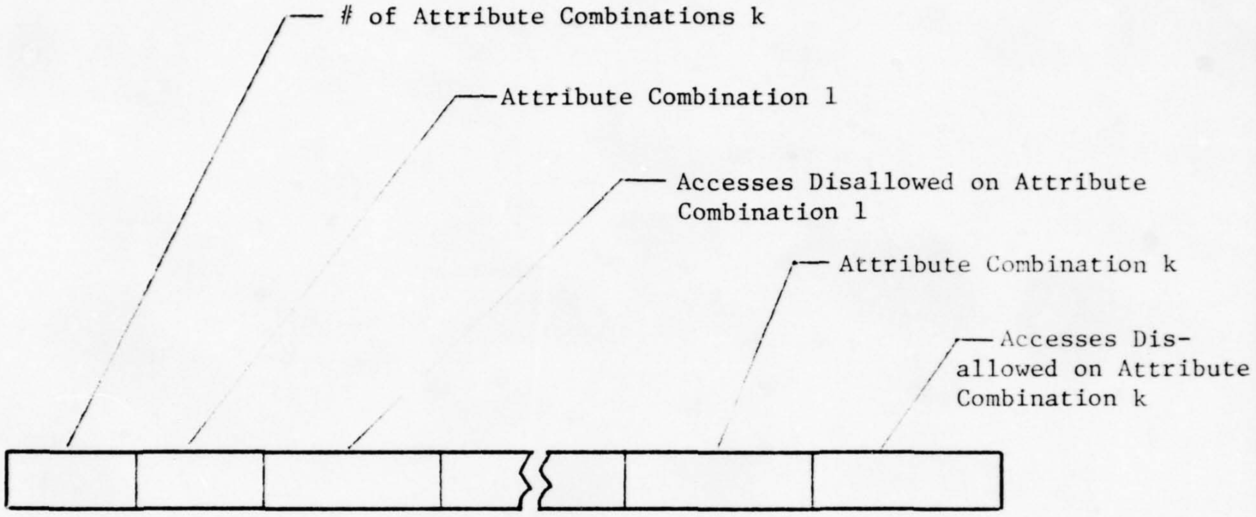
FIGURE 9a. A Logical View of the 'Load-Access-Control-Information' Request

access privilege is shown in Figure 9b. These accesses need not be on entire records, but may be only on certain combinations of attributes in the records. Thus, for example, the creator of the file may specify that user U does not have 'read' access to the Job and Department attributes of all records satisfying security conjunction Si. A default access descriptor is also included as part of this request. This is defined as the set of accesses allowed on all records belonging to the security atom to which no security conjunction is applicable.

The file creator now issues a series of 'Load-record' requests. The format of this request is shown in Figure 10. As the records are received by DBC, two things are determined for each of them, namely, the security atom and the cluster to which it belongs. The cluster that the record belongs to is used in order to determine where in the mass memory this record is to be placed, and also to create a list of cluster identifiers and their descriptions in terms of the primary clustering descriptor and the most important secondary clustering descriptor. The security atom to which the record belongs is used to create a list of security atom identifiers and their descriptions in terms of the security conjunctions making up the atom.

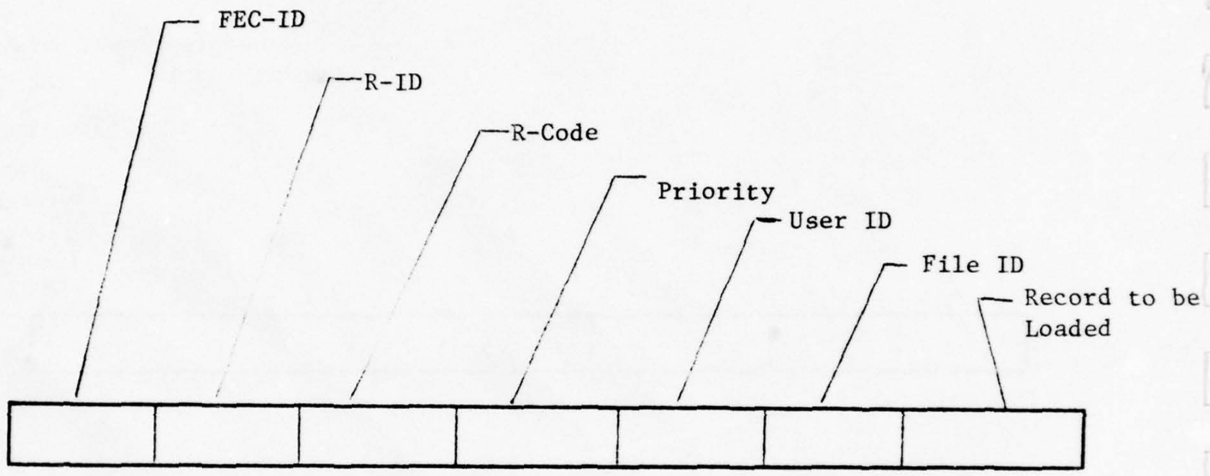
The file creator now issues a 'Close-database-file' request. This request has no arguments. At this stage, DBC does the following for each of the P users allowed to access the file that has just been 'closed' by the creator. For each security atom, the set of security conjunctions that are applicable to that atom are known. At this point, the set of accesses not permitted to each user can be computed with respect to each atom by 'anding' (intersecting) the access descriptors of the security conjunctions applicable to the atom. Thus, the list of atoms and the disallowed accesses is compiled for each of the P users in an atomic access privilege list (AAPL) (one for each user). Since the number of security atoms is expected to be much smaller than the number of records, this list will be relatively short.

When a user command is received by DBC, the record specification of the command can be either a query or a record tag. If the specification is a query, DBC infers the atoms referred to by the query by looking up auxiliary information stored in the structure memory. The record tag, on the other hand, contains the name of the security atom which contains the record



# of Attribute Combinations k
Attribute Combination 1
Accesses Disallowed on Attribute Combination 1
Attribute Combination k
Accesses Disallowed on Attribute Combination k

FIGURE 9b. An Expanded View of the Access Privileges Field in the 'Load-Access-Control-Information' Request



R-Code = $04_8$

FIGURE 10. Logical View of a 'Load-Record' Request

referenced by the user request. By scanning the AAPL of the user for security atom(s), DBC can either deny or permit the access.

## 3. DATA ORGANIZATION IN THE STRUCTURE LOOP

Data structures are maintained in DBC for each file and for the users of the files in order to ensure that no unauthorized accesses are allowed on the database stored in the mass memory of DBC. Also, data clustering is achieved by means of tables that maintain cluster definitions and information on space availability in the disk cylinders comprising the mass memory (MM). This section has been divided into three sub-sections. The first sub-section deals with those tables that are needed for security enforcement. The second sub-section describes the tables needed for data clustering. The discussion of the relationship of the tables in these two sub-sections with DBC components will be included in a later section. The final sub-section describes some of the other tables used by DBC to maintain the names of the users and their files. These tables will have to be referred to by DBC in order to support the mechanisms of access control and data clustering. Some of the tables dealt with in the last sub-section are stored in the database command and control processor (DBCCP) and the rest are stored in the index translation unit (IXU).

We may recall, from Section 1, that an entry in the structure memory (SM) essentially consists of a keyword followed by a list of cylinder numbers. More specifically, an entry in SM consists of a keyword followed by a list of pairs known as index terms, where each index term is of the form (cylinder number, security atom name). The cylinder number indicates the cylinder in MM in which a record containing the keyword may be found, and the security atom name indicates the security atom to which the record belongs. The cluster information, on the other hand, is not stored in the structure memory but is mostly stored in the database command and control processor (DBCCP) and the security filter processor (SFP). The keywords, cylinder numbers and security atom names are stored in SM in a coded fixed-length form in order to conserve space and facilitate searching (it is easier to process fixed-length rather than variable-length fields). The tables in IXU are used to convert the coded index terms in SM to their expanded full-length form. For a detailed explanation of the coding and decoding processes, the reader is referred to [5].

## 3.1 Tables for Security Enforcement

There are three tables maintained by DBC in order to enforce access control. There is one security atom definition table (SADT) for each file which is needed for maintaining the definitions of security atoms in terms of security conjunctions. There is a security information table (SIT) for each user-file combination known to DBC. This table provides a list of security conjunctions and corresponding access privileges for each user of a file as indicated by the file creator. Finally, an atomic access privilege list (AAPL) is created by DBC for each user-file combination in order to keep track of the access privileges allowed to the user with respect to each atom of the file.

### A. The Security Atom Definition Table (SADT)

The security atoms of a file are defined in a security atom definition table (SADT). Hence, there is an SADT for each file. A view of this table is shown in Figure 11. There is an entry in the SADT for each atom of a file and each entry has two fields. The first field is a number that identifies a security atom and the second field is a list of security conjunction identifiers that define the security atom. Every security conjunction specified by a file creator is assigned an identifier by DBC. We may recall, from the previous section, that a security atom consists of records which satisfy the same unique set of security conjunctions. These security conjunctions define the security atom. An example of a SADT would be the one below.

| Security Atom Id | Security Conjunction Ids |
|---|---|
| 1 | S1,S5 |
| 2 | S3,S4,S7 |

Another way of representing the same information is as shown below.

| Security Atom Id | Security Conjunction Id |
|---|---|
| 1 | S1 |
| 1 | S5 |
| 2 | S3 |
| 2 | S4 |
| 2 | S7 |

Either representation may be used in DBC.

| SECURITY ATOM ID | SECURITY CONJUNCTION IDS |
|---|---|
| | |
| . | |
| . | |
| . | |
| | |

FIGURE 11: THE SECURITY ATOM DEFINITION TABLE (SADT)

B. The Security Information Table (SIT)

There is one such table for every user of a file. Thus, if there are n files in the database F1, F2, F3, ..., Fn and the file Fi is accessible by Ui users, then the total number of SITs maintained in DBC is U1 + U2 + ... + Un. A logical view of this table is shown in Figure 12. The first field consists of a security conjunction. The second field consists of a set of field-level security descriptors of the form (attribute combination, access descriptor). The access descriptor always specifies accesses which are not allowed on the corresponding attribute combination. Thus, an access descriptor specifies a set of disallowed accesses. It is seen, for example, in Figure 12 that read requests on both the Name and Position fields of records satisfying the security conjunction (Salary $\geq$ 20,000) $\wedge$ (Position=PROFESSOR) will be denied by DBC.

This table may actually be implemented in the manner indicated in Figure 13. The implementation uses one access template per file. This access template consists of a set of pairs of the form (combination identifier: attribute combination). In other words, each attribute combination intended for protection is assigned a combination identifier. The second field of the SIT is also a set of pairs of the form (combination identifier: access descriptor). Thus, only identifiers and not the actual attribute combinations themselves need to be stored in the SITs. Similarly, the first field in the SIT need not contain a security conjunction. It only needs to contain a security conjunction identifier.

C. The Atomic Access Privilege List (AAPL)

There is one such list for every user-file pair. A diagram of the AAPL is shown in Figure 14. The first field contains a security atom identifier, while the second field consists of a set of field-level security descriptors. This field is similar to the second field of the SIT.

The following steps are taken by DBC to determine if access ai is permitted on attribute combination ACi of a record in security atom SA.

(1) Look up the AAPL for the entry whose first field contains the identifier corresponding to security atom SA. Thus, form the set S of all attribute combinations in the second field of this entry that are subsets of ACi.

(2) Take the union of the access descriptors corresponding to every member

| | |
|---|---|
| (SALARY $\geq$ 20,000) $\wedge$ (POSITION = PROFESSOR) | {([NAME, SALARY], NO-MODIFY), ([NAME, POSITION], NO-READ)} |
| ⋮ | ⋮ |
| | |

FIGURE 12. THE SECURITY INFORMATION TABLE (SIT)

<u>ACCESS TEMPLATE</u> {(1: [NAME, POSITION]), (2: [NAME, SALARY])}

| | |
|---|---|
| (SALARY $\geq$ 20,000) $\wedge$ (POSITION = PROFESSOR) | {(1: NO-READ), (2: NO-MODIFY)} |
| . . . | |
| | |

FIGURE 13: ACTUAL IMPLEMENTATION OF THE SIT.

| | |
|---|---|
| 1 | {([NAME, SALARY], NO-MODIFY), ([NAME, POSITION], NO-READ)} |
| . . . | |
| | |

FIGURE 14: THE ATOMIC ACCESS PRIVILEGE LIST (AAPL)

of S. The resultant access descriptor AD gives the list of disallowed accesses on ACi.

(3) Scan AD to see if ai is disallowed. If not, the access ai is permitted on the combination ACi.

Consider that we are given the AAPL of Figure 14. We are required to determine if the 'read' access is allowed on the combination of the Name, Salary and Length-Of-Service fields of records in Atom 1. Scanning through the second field of the first entry in the AAPL, we determine that the combination [Name, Salary] is the only combination which is a subset of the given combination [Name, Salary, Length-Of-Service]. The access descriptor corresponding to [Name, Salary] is 'No-Modify'. Hence, it allows 'read' access. Thus, 'read' access is permitted on the combination of the Name, Salary and Length-Of-Service fields of records in Atom 1.

## 3.2 Tables for Data Clustering

For the purpose of data clustering, information is needed in DBC about space availability in the mass memory and about the definition of the clusters. The space available in each cylinder allocated to a file is displayed in a <u>cylinder space table</u> (CST). The primary and secondary clustering descriptors are defined in a <u>primary clustering descriptor table</u> (PCDT) and a <u>secondary clustering descriptor table</u> (SCDT), respectively. The definition of each cluster of a file, in terms of primary and secondary clustering descriptors, is carried in a <u>cluster identifier definition table</u> (CIDT).

### A. The Cylinder Space Table (CST)

A logical view of this table is shown in Figure 15. There is one such table for every file known to DBC. There are as many entries in the CST as there are cylinders allocated to the file. Each entry consists of two fields. The first field contains a cylinder number and the second field contains information concerning the unused space in that cylinder.

### B. The Cluster Identifier Definition Table (CIDT)

There is one such table for every file known to DBC. Each entry in the table is a quadruple as shown in Figure 16. The first field contains a cluster identifier. The second and third fields contain the identity of the primary clustering descriptor and the secondary clustering descriptor, respec-

| Cylinder Number | Space Available in Cylinder |
|---|---|
| . . . . | |
| | |

Figure 15: The Cylinder Space Table (CST)

Figure 16: The Cluster Identifier Definition Table (CIDT)

| Cluster ID | Primary Clustering Descriptor ID | Secondary Clustering Descriptor ID | Cylinder Number |
|---|---|---|---|
| | . . . | | |
| | | | |

tively, that make up this cluster. The fourth field contains the cylinder number of a cylinder that contains at least one record belonging to this cluster.

C. The Primary Clustering Descriptor Table (PCDT)

The logical view of one such table is shown in Figure 17. There is one primary clustering descriptor table (PCDT) for each file known to DBC. There is an entry in the PCDT for every primary clustering descriptor and each entry consists of five fields as shown in the figure. The first field contains the identifier of the attribute in the primary clustering descriptor. This identifier is assigned by a front-end computer. In case the primary clustering descriptor is of type-A, then the second and third fields contain the upper and lower limits, respectively, of the range that defines the descriptor. If the primary clustering descriptor or sub-descriptor is of type-B or type-C, then the second field is null (indicated by some special symbol such as a $ sign) and the third field stores the value of the descriptor or sub-descriptor. The second and third fields always contain values in their original form rather than the coded form used to store them in SM. The fourth field contains the identifier assigned to the descriptor by DBC. The fifth field contains the maximum space requirement for the clustering descriptor. This maximum space requirement is the estimated number of bytes needed for storing all records whose primary clustering keywords are derivable from the clustering descriptor. Naturally, the file creator has to make this estimation.

D. The Secondary Clustering Descriptor Table (SCDT)

This table is similar to the PCDT. However, it does not have a fifth field since no maximum space requirement is specified for secondary clustering descriptors. Also, there is now one entry per secondary clustering descriptor instead of one entry per primary clustering descriptor.

## 3.3 Other Table Structures

### 3.3.1 Tables in DBCCP

A description of only those tables that would have to be indirectly used in the enforcement of access control or in the process of data clustering are described here. The reader may wish to refer to [6] for details

| ATTRIBUTE ID | LOWER LIMIT | UPPER LIMIT | PRIMARY CLUSTERING DESCRIPTOR ID | MAXIMUM SPACE REQUIREMENT |
|---|---|---|---|---|
| | | ⋮ | | |
| | | | | |

FIGURE 17: THE PRIMARY CLUSTERING DESCRIPTOR TABLE (PCDT)

concerning some of the other tables.

## A. The File Table (FT)

There is only one _file table_ in DBCCP and it has as many entries as there are files known to DBC. The FT is stored in some global location which is known to all components of DBC. Figure 18 provides a diagram of this table. Each entry of the FT consists of two fields. The first is a file identifier and this is a numeric encoding of the file name. The second field is a pointer to the file information table (FIT) which will be described in the following sub-section.

## B. The File Information Table (FIT)

A logical view of the _file information table_ (FIT) is shown in Figure 19. There is one such table for every file known to DBC. Logically, it consists of 16 separate fields defined as follows.

1) This field contains the identity of the file creator.
2) This two bit field indicates the status of the file. A file may be in one of three possible states.
    a) It may be inactive as indicated by ØØ in this field.
    b) It may be open for creation as indicated by Ø1 in this field. In this state, the file is being created, but nobody has begun to access it yet.
    c) It may be open for access as indicated by 1Ø in this field. In this state, the file is being accessed by some user.
3) This field really consists of n subfields, where n is the number of attributes defined for the file by the file creator. We may recall that n is specified in the 'Open-database-file-for-creation' request (see Figure 6). The n attribute identifiers assigned to this file by the front-end computer are stored in the n subfields.
4) This field contains the number of mass memory cylinders needed to store the file as indicated by the file creator.
5) This field contains the number of additional mass memory cylinders that may be required to store the file in case the initial allocation as specified in Field 4 is not enough. The information in Fields 4 and 5 come directly from Arguments 2 and 3 of the 'Open-database-file-

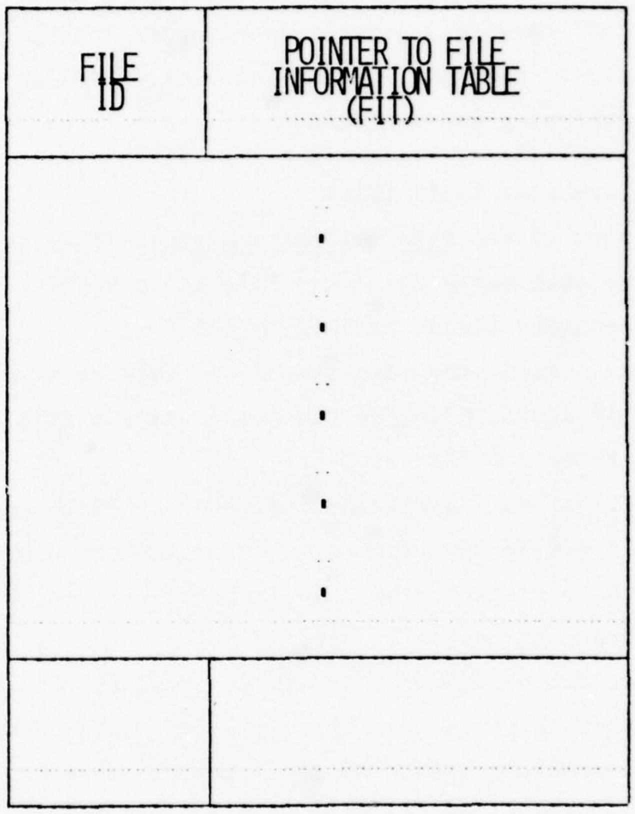


FIGURE 18: THE FILE TABLE (FT)

| |
|---|
| ID of Creator |
| File Status<br>00 File not active<br>01 File open for creation<br>10 File open for access |
| N Attribute ids of the file |
| # of cylinders allocated to the file |
| # of additional cylinders to be allocated |
| Pointer to the CAT in the IXU |
| Pointer to the CBM in the IXU |
| Pointer to the SANBM in the IXU |
| Pointer to the CIBM in the IXU |
| P User ids of users allowed to access this file |
| Pointer to the SADT |
| Pointer to the PCDT |
| Pointer to the SCDT |
| Pointer to the CIDT |
| Pointer to the CST |
| File Identifier |

Figure 19: The File Information Table (FIT)

for-creation' request (see Figure 6).

6) This field contains a pointer to the Cylinder Address Table (CAT) which is stored in the secondary memory of the index translation unit (IXU) and which is described in the last sub-section. The CAT is used to extract absolute cylinder numbers from the index terms retrieved from the structure memory. There is one CAT per file.

7) This field contains a pointer to the Cylinder Bit Map (CBM) which is also stored in the secondary memory of the IXU. There is one CBM (described in the last sub-section) per file. The CBM is used to convert absolute cylinder numbers to relative cylinder numbers before storing them in the structure memory (SM).

8) This field contains a pointer to the Security Atom Name Bit Map (SANBM) stored in the secondary memory of the IXU. There is one SANBM (described in the last sub-section) per file. The SANBM is used to allocate and deallocate security atom identifiers.

9) This field contains a pointer to the Cluster Identifier Bit Map (CIBM) stored in the secondary memory of the IXU. There is one CIBM (described in the last sub-section) per file and it is used to allocate and deallocate cluster identifiers for that file.

10) This field consists of p subfields, where p is the number of users that are allowed to access the file. The value of p is extracted from the 'Load-access-control-information' (see Figure 9) request described in Section 2. The p user identifiers are stored in the p subfields.

11) This field contains a pointer to the SADT for this file.

12) This field contains a pointer to the PCDT for this file.

13) This field contains a pointer to the SCDT for this file.

14) This field contains a pointer to the CIDT for this file.

15) This field contains a pointer to the CST for this file.

16) This field contains the identifier assigned to the file by the front-end computer.

## C. The User Table (UT)

There is only one user table in DBC and it contains one entry for every user known to DBC. The UT is stored in some global location which is known to all the components of DBC. A logical view of the UT is shown in

Figure 2∅. Each entry consists of two fields. The first field contains the identifier of the user, and the second field is a pointer to his user file table (UFT) described in the next sub-section.

D. The User File Table (UFT)

A logical view of the UFT is shown in Figure 21. There is one user file table for each user known to DBC. The UFT is a table of varying length and consists of as many entries as there are files the user is allowed to access. Each entry contains four fields as described below. The first field contains a file identifier. The second field contains a pointer to the SIT for this user corresponding to this file. The third field contains a pointer to the AAPL for this user corresponding to this file. A null entry in the third field indicates that the AAPL for this user-file combination has not yet been created by DBC. The fourth field is a one bit field. A '1' in this field indicates that the user has opened this file for access.

E. The Overall Cylinder Bit Map (OCBM)

There is only one overall cylinder bit map in DBC and it is stored in a known global location. The OCBM has one bit corresponding to each cylinder in the mass memory. If the cylinder is available for allocation, then the corresponding bit in the OCBM contains logic ∅. Otherwise, the bit in the OCBM contains logic 1. DBC uses the OCBM to allocate cylinders to files.

## 3.3.2 Table Structures in IXU

Each file known to DBC has a cylinder address table (CAT) maintained by IXU. A CAT has up to 256 entries, which is the upper limit on the number of cylinders that a file may occupy (this is a decision based on experience with file sizes). There is an entry in a CAT for each cylinder allocated to a file. The CAT is used for transforming relative cylinder numbers (found in the index terms stored in the structure memory) into absolute cylinder numbers. The index terms contain only relative cylinder numbers. A relative cylinder number is used as an index into the CAT to retrieve the corresponding absolute cylinder number from it. Thus, if the relative cylinder number is 3, the third entry in the CAT contains the corresponding

| USER ID | POINTER TO USER FILE TABLE (UFT) |
|---|---|
| . | . |
| . | . |
| . | . |
| | |

FIGURE 20: THE USER TABLE (UT)

FIGURE 21: THE USER FILE TABLE (UFT)

| FILE ID | POINTER TO SIT | POINTER TO AAPL | ONE BIT INDICATES IF USER HAS OPENED FILE FOR ACCESS |
|---|---|---|---|
| | . | | |
| | . | | |
| | . | | |
| | | | |

actual cylinder number.

Corresponding to each CAT there are three bit maps, the Cylinder Bit Map (CBM), the Security Atom Name Bit Map (SANBM), and the Cluster Identifier Bit Map (CIBM). These are used to keep track of the allocation and release of cylinders, security atom names, and cluster identifiers, respectively. For a detailed description of the CAT, the CBM, the SANBM and the CIBM, the reader is referred to [5].

## 4. ALGORITHMS FOR SUPPORTING ACCESS CONTROL AND CLUSTERING

In this section, we present algorithms that are built in the hardware of DBC in order to support the functions of access control and clustering. Some of these algorithms are executed at database creation time, and others are executed at database access time. It is our belief that a better understanding of these algorithms can be obtained if we discuss all the hardware algorithms at database creation time, even if they are not directly related to access control and clustering. Accordingly, we divide this section into two parts, one discussing the algorithms for file creation, and the other discussing the algorithms for file access.

### 4.1 Algorithms for Database Creation

In this sub-section, we present several algorithms, one for each of the preparatory commands mentioned in Section 2. The reader is advised to refer back to the last sub-section of Section 2, in order to look at the nature of these requests again. The algorithms provide a good understanding of how the various tables and lists, described in Section 3, are built up.

ALGORITHM A: To process the command, open-database-file-for-creation.

Step 1: Using the overall cylinder bit map (OCBM), determine if the number of cylinders n requested for initial allocation to the file are available. That is, check if there are at least as many bits with logic Ø in the OCBM as there are cylinders requested for initial allocation. If not, send a reject signal to the front-end computer, and go to Step 4.

Step 2: Allocate a file identifier for the new file. Allocate space in the database command and control processor (DBCCP) for a file information table (FIT) for this file, and make an additional entry in the file table (FT). Fields 1, 2, 3, 4, 5 and 6 of the FIT are filled up at this time. Field 1 is filled with the identity of the file creator. Field 2 is made 'Ø1' to indicate that the file is open for creation. Information for Fields 3, 4 and 5 is obtained from Arguments 1, 2, and 3 of this request (see Figure 6). Field 16 is filled with the identifier assigned to the file by the front-end computer. This information is also present in the request. A cylinder space table (CST) is now created for this file. At this stage, the CST indicates that all n cylinders of the file are empty. Field 15 of the FIT is now filled with a pointer to the CST. Now, a cylinder address table (CAT), a cylinder bit map (CBM), a cluster information bit map (CIBM), and a security atom name bit map (SANBM) are created for this file and loaded into IXU. The CAT contains the absolute cylinder numbers of the n cylinders allocated in Step 1, in its first n locations. The first n bits of the CBM are each filled with logic 1 to indicate that n cylinders have been allocated for the file. Every bit of the CIBM and the SANBM is filled with logic ∅ at the present time. If IXU does not have enough space to hold the CAT, the CIBM, the SANBM and the CBM, then send a reject signal to the front-end computer, and release FIT, FT and CST space allocated to the file. Also, release the cylinders allocated to the file. If the IXU does have enough space, then fill up fields 6, 7, 8, and 9 of the FIT. A check is made to see if the user table (UT) has an entry corresponding to the file creator. If so, then follow the pointer in the entry to the user's user file table (UFT) and go to Step 3. If the UT does not have an entry corresponding to this user, allocate space for an UFT for the user, and then make an entry in the UT for this user. Part of this entry is a pointer to the newly created UFT.

Step 3: Make a new entry in the UFT for this file. In the first field of the UFT entry, enter the file identifier of the file being created. The other fields are left empty at this stage.

Step 4: Release space occupied by the command, and terminate.

ALGORITHM B: To process the command, load-attribute-information.

Step 1: Check, from the second field of the FIT, if the file is open for creation. If the file is not open for creation, send a reject signal to the front-end computer and go to Step 5.

Step 2: Load the hash algorithms of the file into the KXU secondary memory.

Step 3: For each attribute of the file, load the attribute information into the KXU secondary memory.

Step 4: If the KXU has rejected any of the above requests because of lack of space, then send a reject signal to the front-end computer and release FIT, FT, UFT, CST, CAT, CBM, SANBM, CIBM, and UT space allocated to the file in algorithm A. Also, release the cylinders allocated to the file.

Step 5: Deallocate space for the command, and delete the command from DBCCP.

ALGORITHM C: To process the command, load-descriptors.

Step 1: Check, from the FIT, if the file is open for creation. If it is not open for creation, then send a reject signal to the front-end computer and go to Step 6.

Step 2: Allocate space for a primary clustering descriptor table (PCDT) and a secondary clustering descriptor table (SCDT). Fill up Fields 12 and 13 of the FIT with pointers to these two tables, respectively.

Step 3: If the descriptor is a primary clustering descriptor, then an entry is made for it in the PCDT. Otherwise, if the descriptor is a secondary clustering descriptor, then an entry is made for it in the SCDT. If the descriptor is of type-C, then Fields 2 and 3 of its entry are made null (say, represented by a \$ symbol). If the descriptor is of type-B, then Field 2 is null, and Field 3 is filled with the value specified in the descriptor. If the descriptor is of type-A, then the second and third fields are filled with the lower and upper limits, respectively, of the range that defines the descriptor. If the descriptor is of type-C, then Field 4 is left empty at this time. Else, allocate a new descriptor identifier and place it in Field 4 of the entry.

Step 5: Invoke Algorithm C' to form clustering keyword descriptors in the structure memory (SM). We may recall that an entry in SM consists of a keyword followed by a set of index terms. We wish to modify this so that an entry consists of a keyword descriptor followed by a set of index terms. A <u>keyword descriptor</u> is a type-A descriptor and consists of an attribute, a lower limit and an upper limit. That is, it consists of an attribute and a range of values. To find information regarding any keyword Ki in SM, it is only necessary to search the entry corresponding to keyword descriptor KDi, where Ki is derivable from KDi. The advantage of using keyword descriptors is that less space is needed in SM than if keywords were used.

Step 6: Deallocate space for the command, and delete the command from DBCCP.

ALGORITHM C': To form clustering keyword descriptors in the keyword descriptor table (KDT).

Step 1: Using the file identifier passed as argument to this algorithm, search the FT and locate the entry for this file in it. Follow the pointer in the entry to the FIT.

Step 2: Follow the pointer in Field 12 of the FIT to the PCDT for the file. For each type-A descriptor in the PCDT (as indicated by non-null values in Fields 2 and 3 of the entry), do Step 3.

Step 3: Form a keyword descriptor as shown in Figure 22. Store this descriptor in a temporary table called the <u>keyword descriptor table</u> (KDT).

Step 4: Follow the pointer in Field 13 of the FIT to the SCDT for this file. For each type-A descriptor in the SCDT, do Step 3.

Step 5: Terminate.

ALGORITHM D: To process the command, load-access-control-information.

Step 1: Check, from the FIT, if the file is open for creation. If it is not open for creation, then send a reject signal to the front-end computer and go to Step 7.

Step 2: Space is allocated in DBCCP for p SITs, where p is the number of users allowed to access the file. The value of p is part of this command (see Figure 9a. Each SIT will have k entries, where k is the number of security conjunctions specified by the file creator. All the p SITs may now be filled up completely, using the information provided

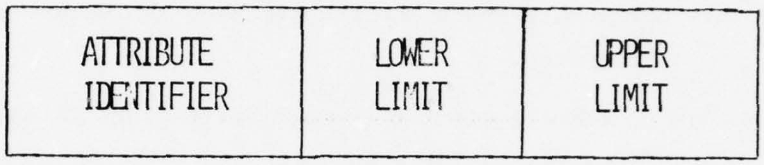


FIGURE 22: FORMAT OF A CLUSTERING OR SECURITY KEYWORD DESCRIPTOR.

by the user in the request. (See Figure 9a.)

Step 3: In the p subfields of Field 1Ø of the FIT, store the p user identifiers, of the p users allowed to access the file. The p user identifiers are part of the information provided by the user with this request.

Step 4: For each of the p users, do Step 5.

Step 5: Check if he already has an entry in the UT. If not, create an entry for him in the UT, and also allocate space for an UFT for him. Now make an additional entry in the UFT. The first field of this entry contains the file identifier. The second field of this entry contains a pointer to the SIT just formed for this user. The third field is null, indicating that the AAPL for this user-file combination has not yet been created. The fourth field is made 'Ø'.

Step 6: Invoke Algorithm D' to form keyword descriptors in SM.

Step 7: Deallocate space for the command, and delete the command from DBCCP.

ALGORITHM D': To form keyword descriptors in the structure memory (SM).

Step 1: Find the entry for the user in the UT. Follow the pointer in the UT to his UFT. Search the UFT for the entry corresponding to this file. Thus, locate the SIT for this user-file combination.

Step 2: Look at the first field of every entry in the SIT. This field contains security conjunctions which are made up of security predicates. We recall that each predicate is a triple of the form (attribute, relational operator, value). For the purposes of this algorithm, we shall only concentrate on the attribute and value parts of a security predicate. Also, we shall restrict ourselves to those security predicates with <u>numeric</u> value parts. For every unique attribute present in these security predicates and in Field 1 of the KDT formed in Algorithm C', do Step 3.

Step 3: Make an entry for this attribute in a temporary table called the descriptor formation table (DFT). Each entry in this table consists of two fields. The first field contains the attribute identifier. The second field contains a list of values. These values are those associated with this attribute in the various security predicates of the file and in the entries in the KDT.

Step 4: For each entry in the DFT, do the following. Remove all duplicates that are present among the values in the second field of this entry. Now, sort the remaining values in the second field of this entry into ascending order.

Step 5: Let each entry be of the form:

```
Attribute ID          V1, V2, ..., Vn.
```

For each entry, form n+1 security keyword descriptors, where each descriptor is of the form shown in Figure 22.

The first descriptor is:

```
Attribute ID      -∞     V1
```

The second descriptor is:

```
Attribute ID      V1     V2
```

The third descriptor is:

```
Attribute ID      V2     V3
```

The n+1th descriptor is:

```
Attribute ID      Vn     ∞
```

Enter these descriptors as new entries in SM. The index terms for these keyword descriptors will be formed as and when records containing keywords derivable from these descriptors are placed in the mass memory (MM) of DBC.

Step 6: Delete the DFT and KDT and terminate.

We illustrate this algorithm in Figures 23, 24, 25 and 26. In Figure 23, we show the four security conjunctions specified by a file creator. Figure 24 shows the descriptor formation table (DFT). Note that the attribute Position has no entry in the DFT, since the values associated with it are non-numeric. In Figure 25, we show the same table after removal of duplicates from the values in all second fields and after sorting the values in all second fields into ascending order. Finally, in Figure 26, we show the security keyword descriptors formed as a result of the security conjunctions specified by the user.

1. $(\text{Salary} > 10{,}000) \wedge (\text{Weight} < 150)$
2. $(\text{Position} = \text{PROFESSOR}) \wedge (\text{Salary} < 2{,}000)$
3. $(\text{Weight} = 130) \wedge (\text{Salary} = 12{,}000)$
4. $(\text{Salary} = 10{,}000)$

Figure 23: Security conjunctions of a file.

| ATTRIBUTE ID | LIST OF ASSOCIATED VALUES |
|---|---|
| SALARY | 10000, 2000, 12000, 10000 |
| WEIGHT | 150, 130 |

FIGURE 24: A VIEW OF THE DESCRIPTOR FORMATION TABLE (DFT).

| ATTRIBUTE ID | LIST OF ASSOCIATED VALUES |
|---|---|
| SALARY | 2000, 10000, 12000 |
| WEIGHT | 130, 150 |

FIGURE 25: A VIEW OF THE DFT, AFTER ELIMINATION OF DUPLICATES AND SORTING.

| ATTRIBUTE ID | LOWER LIMIT | UPPER LIMIT |
|---|---|---|
| SALARY | $-\infty$ | 2,000 |
| SALARY | 2000 | 10,000 |
| SALARY | 10,000 | 12,000 |
| SALARY | 12,000 | $\infty$ |
| WEIGHT | $-\infty$ | 130 |
| WEIGHT | 130 | 150 |
| WEIGHT | 150 | $\infty$ |

FIGURE 26: LIST OF SECURITY KEYWORD DESCRIPTORS FORMED.

ALGORITHM E: To process the command, load-record.

Step 1: Check, from the FIT, if the file is open for creation. If the file is not open for creation, then send a reject signal to the front-end computer and go to Step 8.

Step 2: Invoke Algorithm F to determine the security atom SA to which the record belongs. If the algorithm returns a negative result, then send a reject signal to the front-end computer and go to Step 8.

Step 3: Invoke Algorithm G1 to determine the cluster to which the record belongs. If the algorithm returns a negative result, then send a reject signal to the front-end computer and go to Step 8.

Step 4: Invoke Algorithm G2 to determine the cylinder, in mass memory, in which the record is to be placed. If the algorithm returns a negative result, then send a reject signal to the front-end computer and go to Step 8.

Step 5: Insert the record into the cylinder selected in Step 4.

Step 6: For each directory keyword in the record, do Step 7. We may recall that part of the information provided in the keyword indicates if it is a directory keyword or not.

Step 7: Transform the keyword using the KXU. Using the CBM stored in the IXU, convert the absolute cylinder number obtained in Step 4 into a relative cylinder number. Store the absolute cylinder number in the CAT. Make a new index term, where the index term is defined as given below. The first field contains the relative cylinder number, and the second field contains the security atom identifier SA obtained from Step 2. Search SM to see if the transformed keyword is derivable from any of the keyword descriptors or is one of the keywords in the entries of SM. If so, then enter the index term under this keyword or keyword descriptor. If not, then enter the index term under this new transformed keyword in SM.

Step 8: Deallocate space for the command, and delete the command from DBCCP.

ALGORITHM F: To determine the security atom number for a record for insertion.

Step 1: Let the record for insertion have k keywords. Set i=1, p=1, j=1, n=1.

Step 2: Look at the SIT for this user-file combination. Let there be s entries in the SIT, indicating s security conjunctions. Let the ith security conjunction have Pi predicates. Recall that a security conjunction is a conjunction of security predicates.

Step 3: Look at the first field of the ith entry in the SIT. This field contains the ith security conjunction. Look at the pth predicate of this security conjunction and see if the jth keyword of the record satisfies this predicate. If it does, then go to Step 5. Else, go to Step 4.

Step 4: j=j+1. If j>k, then go to Step 6. Else, go to Step 3.

Step 5: p=p+1. If p>Pi, then go to Step 7. Else, set j=1. Go to Step 3.

Step 6: [The record does not satisfy the ith. security conjunction.]. i=i+1. If i>s, then go to Step 8. Else, set j=1, set p=1, go to Step 3.

Step 7: All the predicates of the ith security conjunction have been satisfied. LIST [n]=i. n=n+1. i=i+1. If i>s, then go to Step 8. Else, set j=1, set p=1, go to Step 3.

Step 8: [LIST[1] through LIST [n-1] contain the set of security conjunction identifiers that describe the security atom to which the record to be inserted belongs. If n=1, then it signifies that no security conjunction defines the atom to which the record belongs, and this is security atom Ø.] If a security atom definition table (SADT) already exists for this file, then search it, looking for an entry that contains LIST[1] through LIST [n-1] in Field 2. If a matching entry is found, then return the atom identifier in Field 1 of the matching entry and terminate. If no such entry exists, or if the SADT itself does not exist, allocate a new security atom name for the file using the SANBM in the IXU. If a new identifier cannot be allocated, return a negative signal and terminate. Else, make a new entry in the SADT. The two fields of this entry are defined below. Field 1 will contain the atom identifier just assigned, and Field 2 will contain LIST [1] through LIST [n-1]. If the SADT has just been allocated, then enter a pointer to the SADT in Field 11 of the SIT.

ALGORITHM G1: To determine the cluster which should contain a record for insertion.

Step 1: Let the record for insertion have k keywords. Let the attribute identifier of the ith keyword be called Ai and the value of the ith keyword be Vi. Set i=1, FLAG=∅, PRIORITY=∅, LIST[1]=∅, LIST[2]=∅, FLAG1=∅, FLAG2=∅, MAX=∅. [FLAG is used to indicate whether the primary clustering keyword for the record has been found. FLAG1 indicates when a new cluster identifier has to be assigned to the file. LIST[1] and LIST[2] will contain, eventually, the primary clustering descriptor identifier and the secondary clustering descriptor identifier from which the primary clustering keyword of the record and the most important secondary clustering keyword of the record are derived.]

Step 2: Search the first field of the PCDT for this file, looking for a match with Ai. Call the set of entries in the PCDT with a matching attribute PC. Let there be n elements in PC. If n is zero, then go to Step 3. Else, go to Step 4. [Here, we are checking to see if the keyword is the primary clustering keyword.]

Step 3: Search the first field of the SCDT for this file, looking for a match with Ai. Call the set of entries in the SCDT with a matching attribute SC. Let there be m elements in SC. If m is zero, then go to Step 18. Else, go to Step 11. [Here, we are checking to see if the keyword is a secondary clustering keyword.]

Step 4: j=1. [In Step 5, we examine the jth element of PC.]

Step 5: Examine the second field of the jth matching entry in the PCDT. If the second field is null, and the third field is not, then compare Vi with the third field. If neither the second nor the third field is null, then compare Vi with the second and third fields, and check to see if Vi lies in between the values specified in the second and third fields. If the comparison is successful, then go to Step 7. If the comparison is unsuccessful, then go to Step 6. If both the second and third fields are null, then set FLAG2 =1. Set MAX=value in fifth field, and go to Step 6.

Step 6: [Check the next element of PC]. j=j+1. If $j \leq n$, then go to Step 5. Else, go to Step 8.

Step 7: Store the fourth field of the matching entry in LIST[1]. FLAG2=Ø. MAX=Ø. FLAG=1. Go to Step 9. [Primary clustering descriptor describing the primary clustering keyword of the record has now been found.]

Step 8: If FLAG2=Ø, then go to Step 9. Else, go to Step 1Ø. [Here, we are checking to see if there is a type-C descriptor with this attribute. If there is, we need to allocate a new type-C sub-descriptor.]

Step 9: i=i+1. If i>k, then go to Step 18. Else, if FLAG=Ø, then go to Step 2. If $i \leq k$ and FLAG=1, go to Step 3.

Step 1Ø: Allocate a new primary clustering descriptor identifier for the file. Make a new entry in the PCDT for this file, where the four fields of this new entry are described below. Field 1 will contain Ai. Field 2 will be null. Field 3 will contain Vi, and Field 4 will contain the new primary clustering descriptor identifier just allocated for the file. Field 5 will contain MAX. This new identifier will also be stored in LIST[1]. FLAG2-=Ø. MAX=Ø. FLAG=1. Go to Step 9. [Primary clustering descriptor describing the primary clustering keyword has been found.]

Step 11: Compare PRIORITY with the relative importance of the attribute Ai as stored in the KXU. We may recall that the relative importance of a secondary clustering attribute is part of the information contained in a 'Load-attribute-information' request (see Figure 7). If PRIORITY is less than the relative importance of Ai, then go to Step 12. Else, go to Step 9.

Step 12: j=1. [In Step 13, we examine the jth matching entry in the SCDT.]

Step 13: Examine the second field of the jth element of SC. If the second field is null, and the third field is not, then compare Vi with the third field. If neither the second nor the third field is null, then compare Vi with the second and third fields, and check to see if Vi lies in between the values specified in the second and third fields. If the comparison is successful, then go to Step 15. If the comparison is unsuccessful, then go to Step 14. If both the second and third fields are null, then set FLAG2 =1 and go to Step 14.

Step 14: j=j+1. If $j \leq m$, then go to Step 13. Else, go to Step 16.

Step 15: Store the fourth field of the matching entry in LIST[2]. FLAG2=∅. PRIORITY=Relative importance of $A_i$ as found from KXU. Go to Step 9. [A secondary clustering descriptor describing a secondary clustering keyword has been found.] Go to Step 9. [A secondary clustering descriptor describing a secondary clustering keyword has been found.]

Step 16: If FLAG2 = ∅, then go to Step 9. Else, go to Step 17. [Here, we check to see if a type-C descriptor with attribute Ai is present. If it is, we need to allocate a new type-C sub-descriptor.]

Step 17: Allocate a new secondary clustering descriptor identifier for the file. Make a new entry in the SCDT for this file, where the four fields of this new entry are described below. Field 1 will contain Ai. Field 2 will be null. Field 3 will contain Vi, and Field 4 will contain the new secondary clustering descriptor identifier just allocated for the file. The new secondary clustering descriptor identifier will also be stored in LIST[2]. FLAG2=∅. PRIORITY=Relative importance of Ai as found from KXU. Go to Step 9.

Step 18: [LIST[1] and LIST[2] contain the primary clustering descriptor and the secondary clustering descriptor, respectively, that describe the cluster to which the record to be inserted belongs. If FLAG=∅, then it indicates that no primary clustering descriptor that may be used for clustering has been found. In this case, we return a negative signal and terminate.] If a cluster identifier definition table (CIDT) exists, then, search the CIDT of this file to determine if an entry for a cluster which may be described by the primary clustering descriptor identifier in LIST[1] and the secondary clustering descriptor identifier in LIST[2] exists.

If such an entry does exist, then set CLUSTER to be equal to Field 1 of this entry, and terminate. Otherwise, set FLAG1=1, and allocate a new cluster identifier for the file using the CIBM described in the previous section. If a new cluster identifier cannot be allocated, return a negative signal and terminate. Else, set CLUSTER to be equal to the new identifier just allocated.

Step 19: Return CLUSTER and terminate.

ALGORITHM G2: To select a cylinder into which a given record may be inserted.

Step Ø1: If FLAG1=1, go to Step 1Ø. Else, go to Step 2.

Step Ø2: Search the CIDT, looking for all entries whose second field contains LIST[1] and whose third field contains LIST[2]. Extract the fourth field from all such entries and put them in a set w. For every unique member of w, do Step 3.

Step Ø3: Search the CST of this file, looking for an entry whose first field matches the element in w. Place the matching CST entry (which is a pair) into a set h.

Step Ø4: Every element of h is a pair. Find that element of h whose second part is the largest of all elements in h. If more than one such element exists, then choose one out of these arbitrarily. Let this be the element (a, b).

Step Ø5: Compare b with the length of the record (lr) to be inserted. We may recall, that the length of the record is part of the information contained in the 'Load-record' request (see Figure 1Ø). If $b \geq lr$, then change the CST entry from (a, b) to (a, b-lr), set CYLINDER = a, and go to Step 12. Else, go to Step 6.

Step Ø6: Search the CIDT, looking for all entries whose second field contains LIST[1]. Extract the fourth field from all such entries and put them together in a set w'. Compare the number of unique elements in w' (:w':), with the estimated number of cylinders br needed for storing all records whose primary clustering keyword is derivable from the primary clustering descriptor in LIST[1]. This number may be found from the fifth field of the entry in the PCDT, whose fourth field matches LIST[1]. If :w': < br, then go to Step 1Ø. Else, for every unique member of w', do Step 7.

Step Ø7: Search the CST of this file, looking for an entry whose first field matches the element in w'. Place the matching CST entry (which is a pair) into a set h'.

Step Ø8: Find that element of h', whose second part is the largest of all elements in h'. If more than one such element exists, then choose one of them arbitrarily. Let this be the element (a', b').

Step Ø9: Compare b' with lr. If b'≥lr, then change the CST entry from (a', b') to (a', b'-lr), set CYLINDER = a', and go to Step 11. Else, go to Step 1Ø.

Step 1Ø: Search the CST of the file, looking for the entry with the largest number of bytes in the second field. If more than one such entry exists, then choose one of them arbitrarily. Let this be the entry (a", b"). Compare b" with lr. If b"≥lr, then change the CST entry from (a", b") to (a", b"-lr), set CYLINDER = a", and go to Step 11. Else, terminate with a negative signal.

Step 11: Make a new entry in the CIDT, where the four fields of this entry are defined as follows. Field 1 will contain the value in CLUSTER. Field 2 will contain the primary clustering descriptor identifier in LIST[1]. Field 3 will contain the secondary clustering descriptor identifier in LIST[2]. Field 4 will contain the value in CYLINDER.

Step 12: Return CYLINDER and terminate. [Note: This algorithm makes use of CLUSTER and FLAG1 from the previous algorithm.]

ALGORITHM H: To process the command, close-database-file.

Step Ø1: Check to see if the file is open, whether for access or for creation. If it is already inactive, then send a reject signal to the front-end computer and go to Step 3.

Step Ø2: If the file is open for access, then look at the entry for this file in the user's UFT and change the fourth field to logic 'Ø'. Also, make the necessary changes in the second field of the FIT to indicate that the file is inactive.

Step Ø3: Deallocate space for the command, and delete the command from DBCCP.

## 4.2 Algorithms for database access

This sub-section deals with the two most common kinds of requests that are made upon any database. These are the 'Insert-record' and the 'retrieve-by-query' requests. The algorithm executed upon receipt of a 'delete-file' request is also included in order to emphasize that this command is different from the 'close-database-file' command.

ALGORITHM I: To process the command, open-database-file-for-access.

Step 1: Check if the file is open for creation. If so, then send a reject signal to the front-end computer and go to Step 6.

Step 2: Check to see if there is an entry for the user in the UT. If not, then send a reject signal to the front-end computer, and go to Step 6. If he does have an entry, then follow the pointer in the UT entry to his UFT.

Step 3: Search the UFT, looking for the entry corresponding to this file. The fourth field of the UFT is made '1' to indicate that the user has opened this file for access. If the third field of this entry is null, then go to Step 4. Else, go to Step 5.

Step 4: For each security atom found in the SADT, invoke Algorithm J to determine the access privileges accorded to the security atom. Thus, form the AAPL for this user-file combination. Change Field 3 of the UFT entry to point to the AAPL just formed.

Step 5: Change Field 2 of the FIT to indicate that the file is open for access.

Step 6: Deallocate space for the command, and delete the command from DBCCP.

ALGORITHM J: To determine the access privileges accorded to a security atom.

Step 1: Search the SADT, looking for an entry whose first field contains the given security atom identifier sa. Form the set SD, consisting of all security conjunction identifiers which define the argument security atom and which are stored in Field 2 of the matching entry. Also, form a field-level security descriptor which allows all accesses on all attribute combinations and store this in AD.

Step 2: If SD is non-empty, then, for every element sd in SD, do Step 3. Else, set AD=AD $\wedge$ Default access descriptor and go to Step 4. We recall that the default access descriptor is part of the 'Load-access-control-information' command (see Figure 9a).

Step 3: Going through the SIT of this user-file combination, find that entry that has sd in the first field. Let Ai be the corresponding field-level security descriptors in the second field of this entry. Set AD=AD $\wedge$ Ai. [Notice that both AD and Ai are sets of field-level security descriptors. We are required to intersect these two sets and come up with a new set of security descriptors. We proceed in the following manner. We form the set S consisting of every field-level security descriptor that appears in either of the two sets. Now, for each field-level security descriptor Di in S, we do the following. Recall that a field-level security descriptor consists of an attribute combination part ACi and an access descriptor part ADi. Going through the remaining field-level security descriptors in S we come up with the set P of field-level security descriptors each of whose attribute combination part is a subset of ACi. Now, we look through the access descriptor part of every member of P to see if there is at least one element in P which has an access descriptor part that disallows all accesses disallowed by ADi. [Recall that an access descriptor is a list of disallowed accesses.]. If at least one such element exists, then delete Di from S. After this process has been repeated for every field-level security descriptor in S, the remaining set S is the required intersected set.]

Step 4: Make a new entry in the AAPL. The two fields of this new entry are defined below. Field 1 will contain the security atom identifier sa. Field 2 will contain the set of field-level security descriptors AD. Terminate.

ALGORITHM K: To process the delete-file command.

Step 1: Check the FIT to see if the file is open for access. If so, then send a reject signal to the front-end computer and to Step 9. [A delete-file command may be issued only fc lowing a close-database-file request.]

Step 2: Check to see if an entry exists for the user in the UT. not, then send a reject signal to the front-end compute and go to Step 9. If he does have an entry, then follow the pointer in the UT entry to his UFT.

Step 3: Search the UFT, looking for the entry corresponding to t file. Using this entry, search the AAPL for this user-f combination to determine if the delete-file access is al [Note: accesses such as delete-file, which are applicab the file level, and not at the security atom level, may checked for by looking at the access privileges correspo ing to any one security atom and any one attribute combi tion. The delete-file access will either be uniformly allowed, or uniformly declined for every (security atom, attribute combination) pair.] If it is not allowed, the send a reject signal to the front-end computer, and go t Step 9. If the delete-file privilege is allowed, then d the AAPL and the entry in the UFT. If it happens to be only entry in the UFT, then delete the UT of this user.

Step 4: Delete attribute information in the IXU for each attribu identifier allocated to the file as indicated in the FIT

Step 5: Release all cylinders and all primary clustering and sec dary clustering identifiers allocated to the file.

Step 6: Release space occupied by the SADT, all the SIT's of thi file, all the AAPL's of this file, the CST, the CIDT, th PCDT, the SCDT, the CAT, the CBM, the CIBM, the SANBM, a the only entry in the FT. For every user allowed to acc the file as indicated in Field 1Ø of the FIT, do Step 7.

Step 7: Delete the entry corresponding to this file from his UFT If this entry is the only one in the UFT, then also dele the UT entry for this user.

Step 8: Release the space occupied by the FIT.

Step 9: Deallocate space for the command, and delete the command from DBCCP.

ALGORITHM L: To process the insert-record command.

Step 1: Follow the UT entry of this user to his UFT. If there i no entry for him in the UT, then send a reject signal to the front-end computer, and go to Step 8. Else, look at that entry in the UFT corresponding to this file. If the fourth field of this entry is not 1, then send a reject signal to the front-end computer, and go to Step 8. [Note: In this step, we check to see if the user has opened this file for access.]

Step 2: Invoke Algorithm F to determine the security atom SA to which the record belongs. If the algorithm returns a negative result, then send a reject signal to the front-end computer, and go to Step 8.

Step 3: Look up the AAPL of this user-file combination, to see if the insert access is permitted on all the attributes of records in atom SA. That is, check if the insert access is denied on any attribute combination. If it is denied, then send a reject signal to the front-end computer, and go to Step 8.

Step 4: Invoke Algorithm G1 to determine the cluster to which the record belongs, and Algorithm G2 to determine the cylinder in which the record is to be placed. If either algorithm returns a negative result, then send a reject signal to the front-end computer, and go to Step 8.

Step 5: Insert the record into the cylinder selected in Step 4.

Step 6: For each directory keyword in the record, do Step 7.

Step 7: Transform the keyword using the KXU. Using the CBM stored in the IXU, convert the absolute cylinder number obtained in Step 4 into a relative cylinder number. Store the absolute cylinder number in the CAT. Make a new index term, where the entry is as defined below. The first field contains the relative cylinder number, and the second field contains the security atom identifier SA obtained in Step 2. Search SM to see if the transformed keyword is derivable from any of the keyword descriptors or is one of the keywords in the entries of SM. If so, then enter the index term under this keyword or keyword descriptor. If not, then enter the index term under this new transformed keyword in SM.

Step 8: Deallocate space for the command, and delete the command from DBCCP.

ALGORITHM M: To process the command, retrieve-by-query.

Step 1: Follow the UT entry of the user to his UFT. If there is no entry for him in the UT, then send a reject signal to the front-end computer, and go to Step 8. Look at the entry in the UFT corresponding to this file. If the fourth field of this entry is not 1, then send a reject signal to the front-end computer, and go to Step 8. [Note: In this step, we check to see if the user has opened this file for access.]

Step 2: We may recall, that a query consists of a disjunction of conjunctions, where each query conjunction consists of a conjunction of keyword predicates. Index terms, from SM, for a query are determined by determining the index terms for each query conjunction. The index terms for a query conjunction are determined by finding the set of index terms Si, for every keyword predicate Ki in the conjunction, and intersecting all the sets thus formed. For exact details regarding how the index terms are obtained and intersected, the reader is referred to [5]. The set of index terms is stored in a set I.

Step 3: We may recall, that each index term consists of a pair, where the first part contains a relative cylinder number, and the second part contains a security atom identifier. Form the set SA which consists of the second part of every element in I. That is, extract all security atom identifiers from index terms. Once again, the reader is referred to [5] for details of how this is done in the IXU.

Step 4: If SA is non-empty, then, for each member sa of SA, do Step 5.

Step 5: Search the AAPL for this user-file combination, looking for an entry whose first field is sa. If no such entry exists, then invoke Algorithm J to determine the access privileges for this atom and to make a new entry in the AAPL for this atom. Search the second field of this entry to determine if the retrieve (or read) access is denied on the requested attribute combination or any subset of it (the user need not ask for entire records in a 'retrieve-by-query' request.). If it is denied, then delete the element i in set I from which sa was derived.

Step 6: For each member i of I, do Step 7. If I is empty, then send a reject signal to the front-end computer and go to Step 8.

Step 7: Convert the relative cylinder number i1, which is the first part of i, to an absolute cylinder number AC, using the CAT. Let i2 be the security atom identifier stored in the second part of i. Now issue the request to the mass memory to retrieve, from cylinder AC, all records (or parts of records, depending on the attribute combination specified by the user) which satisfy the given query and which are part of security atom i2.

Step 8: Deallocate space for the command, and delete the command from DBCCP.

## 5. AN ILLUSTRATION OF THE SECURITY AND CLUSTERING MECHANISMS

In this section, we shall try to provide a clearer understanding of the security and clustering processes described formally in earlier sections, by means of an example. Since the sample database of the example must be small, the reader should not be surprised that the security atoms and clusters formed are necessarily very small. However, it is hoped that the example should serve the useful purpose of clarifying the details of the security and clustering mechanisms. The section is divided into two sub-sections. In the first of these, we illustrate the construction of the various tables and lists during database creation time. In the second sub-section, we illustrate how DBC deals with queries during database access time.

We consider, for illustration, a relational database with two relations EMPLOYEE (Name, Sal, Los, Pos, Ht) and POSITION (Pos, Ma, Mq), where the column names, i.e., attributes of each relation, are enclosed in parenthesis. Every employee record, i.e., a row in relation EMPLOYEE, consists of an employee Name, an employee salary Sal, an employee's length of service Los, a job position Pos and an employee hometown Ht. Every position record, i.e., a row in relation POSITION, consists of a job position Pos, a minimum age Ma and a minimum qualification Mq. The relational database is shown in Figures 27a and 27b.

### 5.1 Table and List Construction

In order to keep the discussion simple, the assumption has been made that a cylinder can accomodate up to five records (instead of some realistic size, such as 500,000 bytes). The creator U of a file ALPHA begins by issuing the 'open-database-file-for-creation' request. In this request, he specifies that file ALPHA has eight attributes, that it will occupy not more than six cylinders initially, and that it may be expanded later on to two additional cylinders. DBC executes Algorithm A in response to this request.

Figure 28a shows a view of the file information table (FIT) after the execution of Algorithm A. The first field indicates that the file has been created by the file creator who has an identifier of 1. The second field in the FIT indicates that the file has been opened for creation. From Field 3, we see that eight attributes have been assigned to the file. The identifiers of these attributes are 1 through 8. Fields 4 and 5 of the FIT indicate, respectively, that six cylinders have been allocated to the file and that two more cylinders may be allocated to it if and when the need arises. The cylinder address table (CAT), the cylinder bit map (CBM), the security atom name bit map (SANBM), the cluster identifier bit map (CIBM) and the cylinder space

FILE ALPHA

EMPLOYEE RELATION

| | Name | Sal | Los | Pos | HT |
|---|---|---|---|---|---|
| R1. | Menon | 20,000 | 6 | Asst. Prof. | Bombay |
| R2. | Hsiao | 20,000 | 12 | Prof. | Taipei |
| R3. | Kerr | 17,000 | 12 | Lecturer | Columbus |
| R4. | Lorenzo | 13,000 | 6 | Junior Lecturer | Mexico City |
| R5. | Jay | 24,000 | 6 | Asst. Prof. | Calcutta |
| R6. | Lalit | 2,000 | 6 | Student | Bombay |
| R7. | John | 37,000 | 12 | Prof. | Baltimore |
| R8. | Jack | 16,000 | 9 | Lecturer | New York |
| R9. | Paul | 18,000 | 9 | Lecturer | Los Angeles |
| R10. | Mark | 12,000 | 6 | Junior Lecturer | Goshen |
| R11. | Jeff | 21,000 | 6 | Asst. Prof. | Iowa |
| R12. | Chris | 24,000 | 6 | Asst. Prof. | Iowa |
| R13. | Pramod | 22,000 | 6 | Asst. Prof. | Lucknow |
| R14. | Munshi | 49,000 | 12 | Prof. | Kanpur |

FILE CREATOR: U

FIGURE 27a. Relation EMPLOYEE of Sample Database

FILE: ALPHA

CREATOR: U

POSITION RELATION

| | Pos | Ma | Mq |
|---|---|---|---|
| R15. | Prof. | 45 | Ph.D. |
| R16. | Asst. Prof. | 40 | Ph.D. |
| R17. | Lecturer | 35 | M.S. |
| R18. | Junior Lecturer | 30 | B.S. |
| R19. | Student | 20 | High School Degree |

FIGURE 27b. Relation POSITION of Sample Database

table (CST) are also created at this time, and pointers to these are entered in Fields 6, 7, 8, 9 and 15, respectively. Field 16 indicates that the file that is being created has the identifier 1.

Figure 28b shows the cylinder space table (CST) where information regarding the availability of space in the cylinders allocated to the file is stored. We see, from the CST, that the cylinders allocated to the file are 1 through 6. Since, at this stage, file ALPHA has not yet been loaded into the mass memory (MM) of DBC, all the six cylinders are empty. This is indicated, in the CST, by the entries in the second fields. All these entries are five, indicating that space is available for five records (which is the maximum capacity of a cylinder) in each of these cylinders.

The user then issues the 'load-attribute-information' request followed by the 'load-descriptors' request. In the former request, the user specifies that the primary clustering attribute of the file is Relation, and the secondary clustering attributes of the file, in order of importance, are Sal and Pos. The primary clustering and secondary clustering descriptors specified by the user in the latter request are shown in Figures 29a and 30a, respectively. The primary clustering descriptor table (PCDT) and the secondary clustering descriptor table (SCDT) formed therefrom by execution of Algorithm C, are shown in Figures 29b and 30b, respectively. We may see, from the last field of the PCDT, that the user expects all the records in the EMPLOYEE relation to fit into four cylinders, and all the records in the POSITION relation to fit into two cylinders. We recall, from our discussion in an earlier section, that if a descriptor is of type-B, then the second field of the corresponding table is made null. This is clearly shown in Figures 29b and 30b. Fields 12 and 13 of the FIT are now filled up with pointers to these two newly created tables. At this point, DBC executes Algoirthm C' to form clustering keyword descriptors and store them in the keyword descriptor table (KDT). We recall that as many clustering keyword descriptors are formed as there are type-A descriptors. Since only one of the four descriptors is of type-A, only one clustering keyword is formed. This is shown in Figure 31.

Next, in a 'load-access-control-information' request, the database creator specifies that four users are allowed to access file ALPHA. He also specifies seven security conjunctions and the access privileges of these four users on records in file ALPHA satisfying these security conjunctions. One of the four users is the creator of the file himself who is identified as 1. Let

| | |
|---|---|
| 1 | 1 |
| 2 | 01 |
| 3 | 1 2 3 4 5 6 7 8 |
| 4 | 6 |
| 5 | 2 |
| 6 | ptr. to CAT |
| 7 | ptr. to CBM |
| 8 | ptr. to SANBM |
| 9 | ptr. to CIBM |
| 10 | |
| 11 | |
| 12 | |
| 13 | |
| 14 | |
| 15 | ptr. to CST |
| 16 | 1 |

FIGURE 28a. A View of the FIT After the 'Open-Database-File-For-Access' Request From User U

| Cylinder | Capacity |
|---|---|
| 1 | 5 |
| 2 | 5 |
| 3 | 5 |
| 4 | 5 |
| 5 | 5 |
| 6 | 5 |

FIGURE 28b. A View of the CST After the 'Open-Database-File-For Access''Request From User U

| Descriptor id | Descriptor | Type of Descriptor |
|---|---|---|
| PCD1. | Relation = EMPLOYEE | Type - B |
| PCD2. | Relation = POSITION | |

FIGURE 29a. The Primary Clustering Descriptors of the File

| Attribute | Lower Limit | Upper Limit | Primary Clustering Descriptor ID | Maximum Space Requirement |
|---|---|---|---|---|
| Relation | $ | EMPLOYEE | 1 | 4 |
| Relation | $ | POSITION | 2 | 2 |

FIGURE 29b. A View of the PCDT After the 'Load-Descriptors' Command

| Descriptor id | Descriptor | Type of Descriptor |
|---|---|---|
| SCD1. | $1,000 \leq Sal \leq 30,000$ | Type-A |
| SCD2. | Position = Asst. Prof. | Type-B |

FIGURE 30a. The Secondary Clustering Descriptors of the File

| Attribute | Lower Limit | Upper Limit | Secondary Clustering Descriptor ID |
|---|---|---|---|
| Sal | 1000 | 30000 | 1 |
| Pos | $ | Asst. Prof. | 2 |

FIGURE 30b. A View of the SCDT After the 'Load-Descriptors' Command

| ATTRIBUTE | LOWER LIMIT | UPPER LIMIT |
|---|---|---|
| Sal | 1,000 | 30,000 |

FIGURE 31. The One Clustering Keyword Descriptor Formed for the File, as Stored in the Keyword Descriptor Table (KDT)

us assume that the other three users have been allocated the identifiers 2, 3 and 4, respectively. DBC now creates four SITs, one for each of these four users. The SIT corresponding to User 2 is shown in Figure 32. The tenth field in the FIT is now filled up to indicate that Users 1, 2, 3 and 4 are allowed to access file ALPHA. Finally, DBC creates security keyword descriptors based on the seven security conjunctions specified by the creator by executing Algorithm D'. These are shown in Figure 33. We also recall that the user must specify a default access descriptor as part of this request. Let us assume that he specifies the default access descriptor to be one which allows all accesses.

The records are now loaded one by one. In representing a relation in the DBC database, a relational tuple is converted to a set of DBC keywords, i.e., attribute-value pairs, and a special keyword is created for the relation name. For example, the first tuple in the EMPLOYEE relation is represented as a DBC record:

(<Relation, EMPLOYEE>,<Name, MENON>,<Sal, 20,000>,
<Los, 6>,<Pos, ASST. PROF>,<Ht, BOMBAY>).

DBC determines, for each record, the security atom and cluster in which it belongs, and the cylinder in which the record is to be placed. While the record is being loaded into the mass memory (MM), the directory in the structure memory (SM) is updated to reflect the presence of the newly inserted record. The directory keywords of a record are specified in the records themselves. Let us assume that the user specifies only security and clustering keywords to be directory keywords.

The security atom definition table (SADT) and the cluster identifier definition table (CIDT) are also formed as the records are being loaded. Let us illustrate the process of loading a record by means of an example. Consider that the first record R1, as shown in Figure 27a, is being loaded. First, the security atom to which the record belongs is determined by identifying those security conjunctions that are satisfied by the record. We shall consider the seven security conjunctions S1, ..., S7 in turn. The first predicate of S1, $(Sal \geq 20,000)$, is satisfied by the keyword <Sal, 20,000>. However, there is no keyword to satisfy the second predicate of S1 which is (Pos=PROFESSOR). Thus, the record does not satisfy S1. However, the record satisfies S2, because the keyword <Sal, 20,000> satisfies the predicate $(Sal \leq 20,000)$ and the keyword <Los, 6> satisfies the predicate (Los=6). The record does not satisfy S3, since it does not contain any keyword to satisfy the first predicate of S3. The record does satisfy S4, because the keyword <Sal, 20,000> satisfies the predicate $(Sal \leq 30,000)$ and the keyword <Los,6> satisfies the predicate (Los=6). The record does not satisfy S5, because no keyword to satisfy the

SECURITY INFORMATION TABLE (SIT)

| Security Conjunction | Set of Access Descriptors |
|---|---|
| S1. $(Sal \leq 20{,}000) \wedge (Pos = Professor)$ | {([Name, Sal], No-Modify), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read)} |
| S2. $(Sal \leq 20{,}000) \wedge (Los = 6)$ | {([Name, Los], No-Modify)} |
| S3. $(Pos = Professor) \wedge (Los = 12)$ | {([Name, Sal], No-Read), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read)} |
| S4. $(Sal \leq 30{,}000) \wedge (Los = 6)$ | {([Name, Sal], No-Read), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read)} |
| S5. $(Sal \leq 30{,}000) \wedge (Los = 9)$ | {([Name, Los], No-Read), ([Name, Pos, Ht], No-Modify)]} |
| S6. $(Sal \leq 1{,}000) \wedge (Los = 12)$ | {([Name, Sal], No-Read), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read)} |
| S7. $(Pos = Professor)$ | {([Name, Sal], No-Read), ([Name, Los], No-Modify), ([Name, Pos, Ht], No-Modify), ([Pos, Ma, Mq], No-Modify)} |

FIGURE 32. The SIT Corresponding to User A (Identifier 2) on File ALpha (File id 1). One of the Four SITs Formed After the 'Load-Access-Control-Information' Request.

| Attribute id | Lower Limit | Upper Limit |
|---|---|---|
| Sal | $-\infty$ | 1000 |
| Sal | 1000 | 20000 |
| Sal | 20000 | 30000 |
| Sal | 30000 | $\infty$ |
| Los | $-\infty$ | 6 |
| Los | 6 | 9 |
| Los | 9 | 12 |
| Los | 12 | $\infty$ |

FIGURE 33. The Security Keyword Descriptors Formed for the File

predicate (Los=9) exists in the record. Similarly, since no keyword to satisfy the predicate (Los-12) exists in the record, S6 is not satisfied by the record. Finally, the record does not satisfy S7 because no keyword to satisfy (Pos= PROFESSOR) exists in the record. Thus, the record satisfies security conjunctions S2 and S4, and is part of Atom 1 which is defined by these two conjunctions. This is reflected in Figure 34.

To determine the cluster to which the record belongs, DBC proceeds as follows. It determines the primary clustering keyword to be <Relation, EMPLOYEE>. This is obviously derivable from the primary clustering descriptor PCD1 which is (Relation=EMPLOYEE). DBC then determines that the record has two secondary clustering keywords <Sal, 20,000> and <Pos, ASST. PROF>. However, the former is more important and is the one that shall be considered for clustering purposes. We see that <Sal, 20,000> is derivable from SCD1 which is ($1,000 \leq Sal \leq 30,000$). Thus, the cluster the record belongs to is defined by PCD1 and SCD1 and this is Cluster 1.

Finally, DBC must determine the cylinder into which the record must be inserted. Since this is the first record being inserted, DBC arbitrarily chooses to insert the record into Cylinder 1.

At the end of the loading process, the state of the security atom definition table (SADT), the cluster identifier definition table (CIDT) and the directory in SM are shown in Figures 34, 35, and 36 respectively. Each directory entry consists of a keyword or keyword descriptor, followed by index terms. Each index term is a pair of the form (cylinder number, atom number).

Figures 37 and 38 clearly show the formation of the security atoms and the clusters in terms of records. Figure 39 indicates which records have gone into which cylinders. The final form of the CST is shown in Figure 40. We see from the CST that Cylinders 1 and 3 are incapable of holding any more records as they are filled to capacity. The final form taken by the FIT is shown in Figure 41. The only field that needs explanation is Field 10 which indicates that four users with identifiers 1, 2, 3 and 4 are allowed to access the file.

## 5.2 Execution of Requests

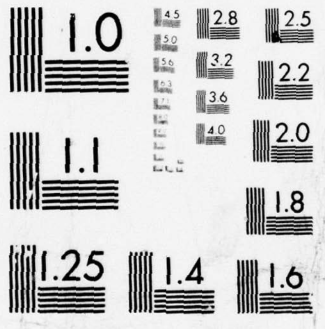
Before a user may issue any access command, he must issue an 'open-database-file-for-access' request. Consider that User 2 issues the above request. At this stage, the atomic access privilege list (AAPL) is created. A view of User 2's AAPL is shown in Figure 42.

We now consider the execution of three typical access requests by the user.

AD-A068 815 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 5/2
THE CLUSTERING AND SECURITY MECHANISMS OF A DATABASE COMPUTER (--ETC(U)
APR 79 J BANERJEE, D K HSIAO, J MENON N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-79-2 NL

2 OF 2
AD
A068815

END
DATE
FILMED
6 --79
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

| Security Atom id | Security Conjunction ids |
|---|---|
| 0 | ƀ |
| 1 | S2, S4 |
| 2 | S1, S3, S6, S7 |
| 3 | S6 |
| 4 | S4 |
| 5 | S5 |
| 6 | S7 |

FIGURE 34. A View of the Security Atom Definition Table After all Records Have Been Loaded (SADT)

| Cluster ID | Primary Clustering Descriptor ID | Secondary Clustering Descriptor ID | Cylinder Number of Cylinder That has at Least 1 Record of This Cluster |
|---|---|---|---|
| 1 | PCD1 | SCD1 | 1 |
| 1 | PCD1 | SCD1 | 3 |
| 1 | PCD1 | SCD1 | 4 |
| 2 | PCD1 | ƀ | 2 |
| 3 | PCD2 | ƀ | 5 |
| 4 | PCD2 | SCD2 | 6 |

FIGURE 35. A View of the Cluster Identifier Definition Table (CIDT) After all the Records Have Been Loaded.

| Keyword or Keyword Descriptor | Index Terms |
|---|---|
| $-\infty \leq$ Sal < 1000 | ---- |
| 1000 $\leq$ Sal < 20000 | (1,1),(3,1) |
| 20000 $\leq$ Sal < 30000 | (1,1),(3,1),(4,1) |
| 30000 $\leq$ Sal < $\infty$ | (2,2) |
| $-\infty \leq$ Los < 6 | ------ |
| 6 $\leq$ Los < 9 | (1,1),(3,1),(4,1) |
| 9 $\leq$ Los < 12 | (3,1) |
| 12 $\leq$ Los | (2,2),(1,1) |
| <Relation, EMPLOYEE> | (1,1),(3,1),(4,1),(2,2) |
| <Relation, POSITION> | (6,4),(5,3) |
| <Pos, PROFESSOR> | (2,2),(5,3) |
| <Pos, ASST. PROF.> | (1,1),(3,1),(4,1) |

FIGURE 36. A View of the Directory in the Structure Memory After all the Records Have Been Loaded

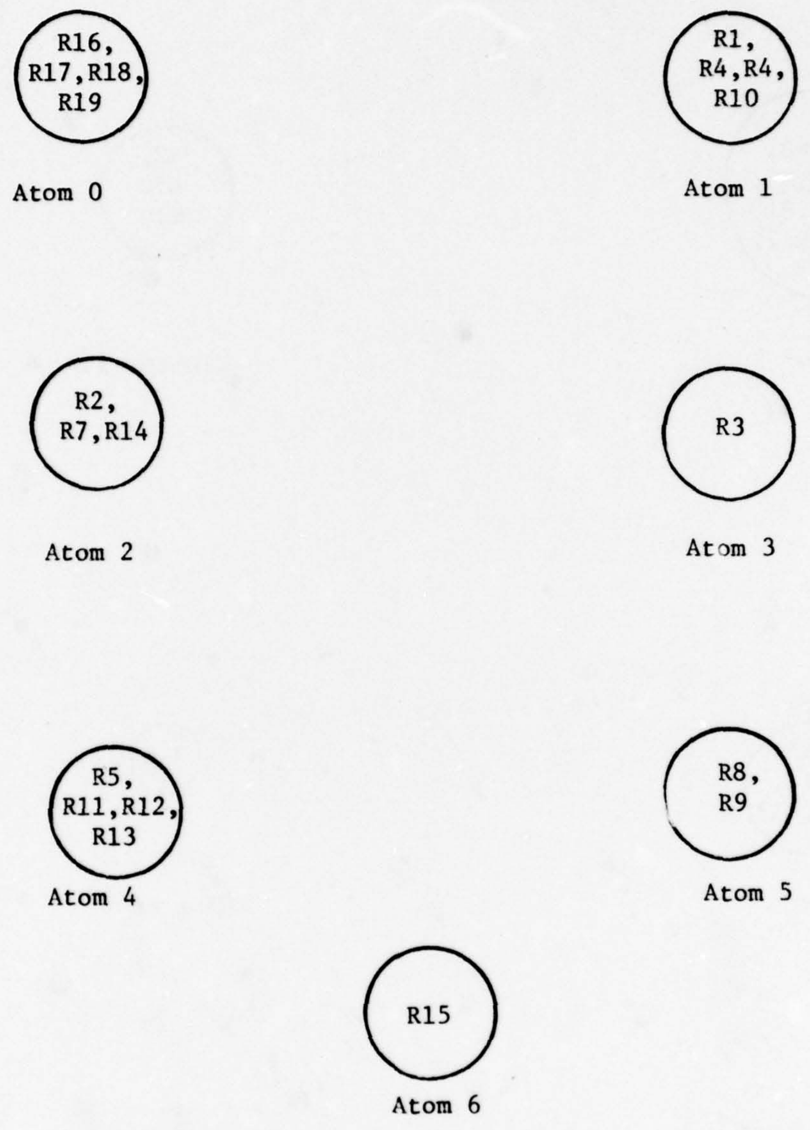


FIGURE 37. A View Showing How Records Fit Into the Seven Atoms

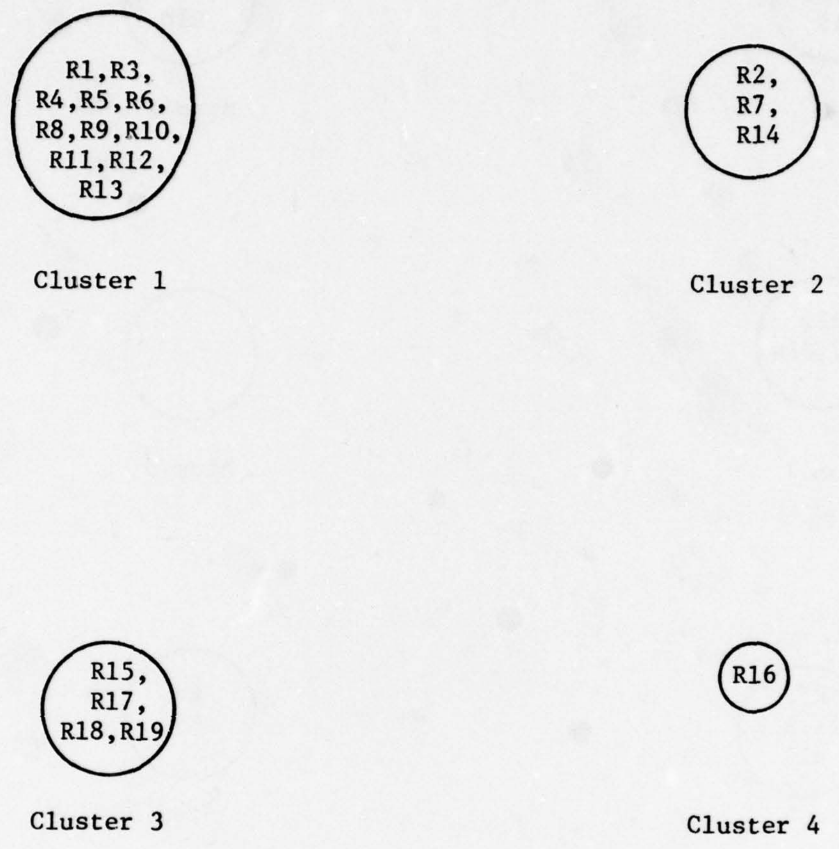


FIGURE 38. A view showing how records are grouped into clusters.

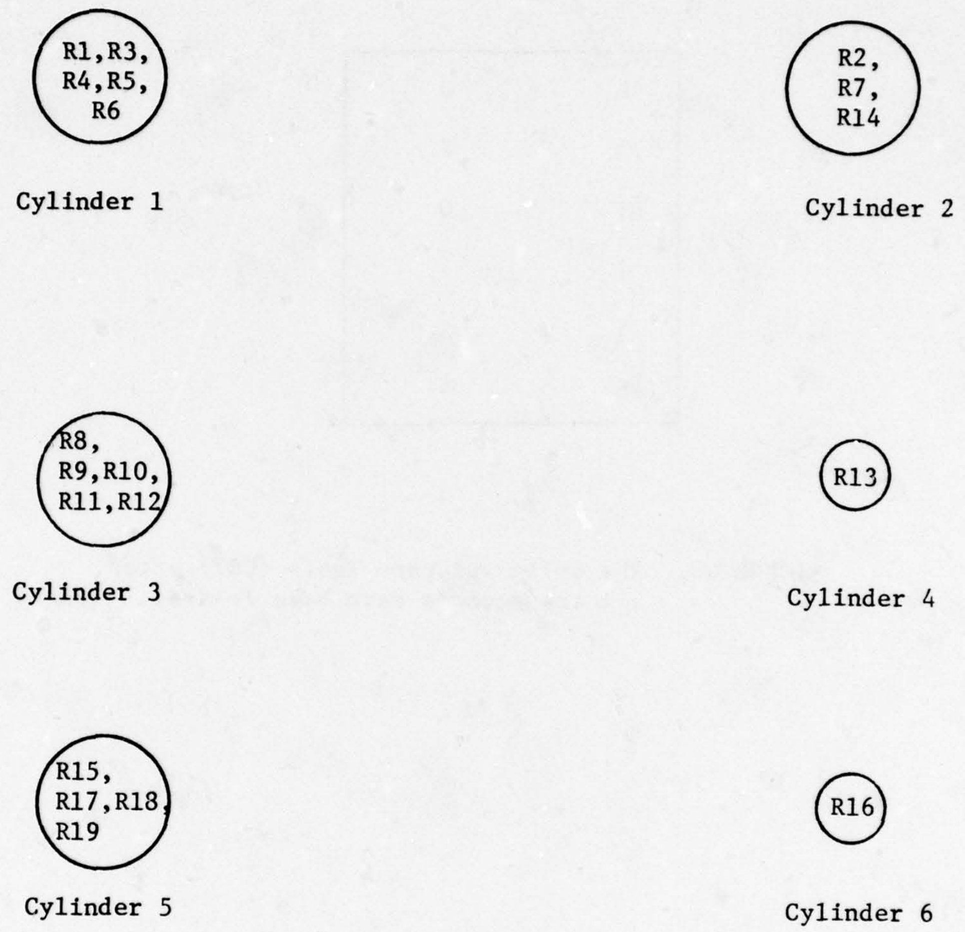


FIGURE 39. A view showing how records are placed in cylinders

| | |
|---|---|
| 1 | 0 |
| 2 | 2 |
| 3 | 0 |
| 4 | 4 |
| 5 | 1 |
| 6 | 4 |

FIGURE 40. The Cylinder Space Table (CST) after all the records have been loaded.

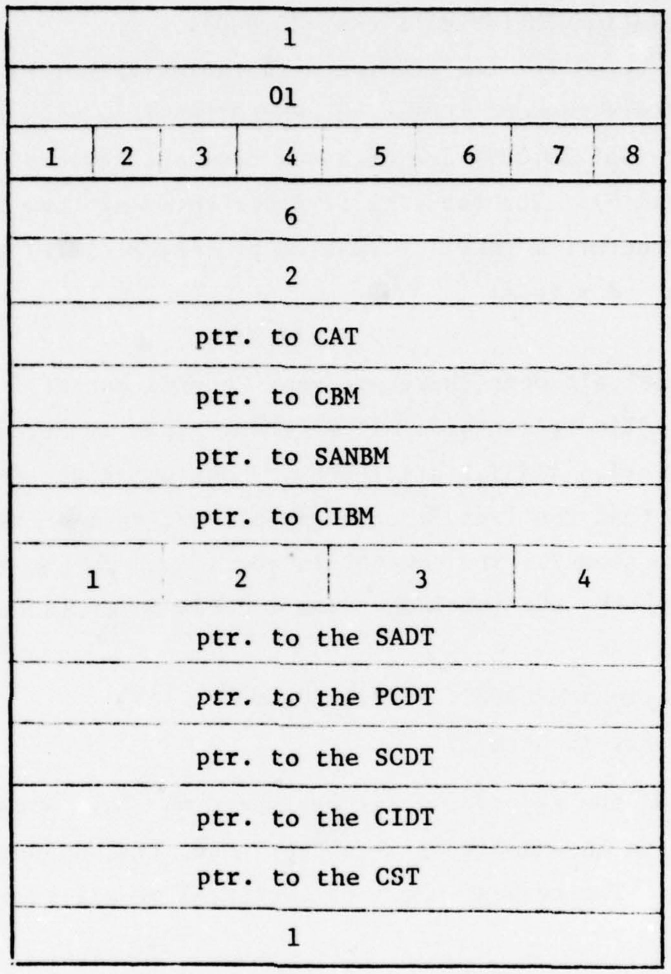


FIGURE 41. A view of the FIT after all records have been loaded.

Example 1: Read the length of service of all employees with salaries greater than $30,000.

This is an example where the physical clustering of the records becomes very useful for efficient execution of the request. The request requires retrieval of all records that satisfy the query

((Relation=EMPLOYEE) & ($Sal \geq 30,000$)).

The index terms for the predicate (Relation=EMPLOYEE) are obtained from the directory keyword <Relation, EMPLOYEE> (see Figure 36). The index terms for ($Sal \geq 30,000$) are obtained from the keyword descriptor ($30,000 \leq Sal < \infty$). The two sets of index terms are now intersected by the structure memory information processor (SMIP) to produce the following index term:

(2,2)

We note that although there are many records satisfying the query, only one cylinder contains all of them. This is because of clustering by the Relation and Sal attributes. Looking at the AAPL for User 2, we notice that the 'read' access is allowed on the Los attribute of records in Atom 2. The request is now issued to the mass memory (MM) to retrieve the Los attribute from records in Cylinder 1, which satisfy the query

(Relation=EMPLOYEE) & ($Sal \geq 30,000$)

and which are in Atom 2.

Example 2: Read the records of all employees who have worked six years.

This is an example of a request where the response set is not clustered. The request requires retrieval of all records that satisfy the query

((Relation=EMPLOYEE) & (Los=6)).

The intersected set of index terms for the query is

(1,1), (3,1), (4,1).

Looking at the AAPL of the user, we note that entire records may not be read from Atom 1. Thus, the request is rejected by DBC.

Example 3: Insert the record

(<Relation, POSITION>, <Pos, SECRETARY>, <Ma, 20>, <Mq, HIGH SCHOOL>)

into file ALPHA.

DBC first determines that the record belongs to security atom 0 since, it does not satisfy any of the security conjunctions. From the AAPL, we see that the accesses disallowed on all attribute combinations of records in Atom 0 are defined in the default access descriptor. Since the default access descriptor allows all accesses, the insert access is permitted. The primary clustering keyword of the record is

AAPL for User A (User Id2)

| Atom ID | Set of Field-Level Security Descriptors |
|---|---|
| 0 | Default access descriptor |
| 1 | {([Name, Sal], No-Read), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read)} |
| 2 | {([Name, Sal], No-Read), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read), ([Pos, Ma, Mq], No-Modify)} |
| 3 | {([Name, Sal], No-Read), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read)} |
| 4 | {([Name, Sal], No-Read), ([Name, Los], No-Read), ([Name, Pos, Ht], No-Read)} |
| 5 | {([Name, Los], No-Read), ([Name, Pos, Ht], No-Modify)} |
| 6 | {([Name, Sal], No-Read), ([Name, Los], No-Modify), ([Name, Pos, Ht], No-Modify), ([Pos, Ma, Mq], No-Modify)} |

FIGURE 42. Atomic Access Privilege List (AAPL) for User A on File Alpha

<Relation, POSITION> and this is derivable from PCD2. Its only secondary clustering keyword <Pos, SECRETARY> is not derivable from any secondary clustering descriptor and is hence assumed to be derived from a null secondary clustering descriptor. Now, it is determined from the CIDT (figure 35), that the record belongs to Cluster 3, and that all records in Cluster 3 are placed in Cylinder 5. The CST (Figure 40) indicates that Cylinder 5 can accommodate one more record. Hence, the request is issued to MM to insert the given record into Cylinder 5.

## 6. A CLOSER LOOK AT THE ARCHITECTURAL ELEMENTS

In this section, we discuss the architectural elements of DBC for access control and data clustering.

Whenever an update command is handled in DBC, we know that the update entails at least a retrieval of data from the mass memory (MM), followed by an insertion of updated data back into MM. We also know that every time a record is inserted into MM, DBC must determine the following three pieces of information about it.

(1) The security atom it belongs to.

(2) The cluster that it belongs to.

(3) The cylinder in which it must be placed.

Obviously, speedy determination of these three types of information will improve the performance of DBC considerably. Such determination is carried out by Algorithms F, G1 and G2 (See Section 4). These algorithms mostly involve the searching of various tables. Naturally, these searches may be speeded up by the use of parallel techniques rather than seuqential ones, and that is what we propose to examine.

Looking at Algorithm F, we see that the security information table (SIT) needs to be searched first. However, the algorithm demands that DBC step through <u>every</u> entry in the SIT. Thus, the table does not lend itself to parallel search hardware. The next table searched by this algorithm is the security atom definition table (SADT). Here, the algorithm looks for a particular set of security conjunction identifiers in the second field of an SADT entry. This is just the kind of search that will be speeded up by the use of parallel search hardware.

Algorithm G1 requires DBC to search the primary clustering definition table (PCDT) and the secondary clustering definition table (SCDT), for entries that have particular values in the attribute fields of these two tables. That is, given an attribute identifier, it is necessary to determine all the entries in PCDT and SCDT that have this identifier as their first field. This search will obviously be speeded up if done in parallel. The algorithm also needs to search the cluster identifier definition table (CIDT), to look for an entry with a particular pair of values in its second and third fields. That is, the primary clustering descriptor identifier PID, and the secondary clustering descriptor identifier SID (from which the primary clustering keyword of a record and the most important secondary clustering keyword of the same record are derivable) are first determined. It is now necessary to search CIDT for an entry with PID in Field 2 and SID in Field 3. This is also the kind of search that may be more speedily done in parallel. Similarly, Algorithm G2 should also search CIDT in a parallel fashion.

The only table searched by Algorithm M (which is executed in response to the 'retrieve-by-query' request) is the atomic access privilege list (AAPL). Given a security atom identifier sa, it is necessary to search AAPL to determine the entry which contains sa in its first field. At first glance, it seems profitable to search elements of AAPL in parallel. However, we notice that the first entry in the AAPL is for Atom 0, the second entry is for Atom 1, and so on. Thus, given Atom sa, it is only necessary to look at the (sa+1)-th entry of AAPL and this can be quite easily done by a single processor with a random access memory.

Thus, we conclude that SADT, CIDT, PCDT and SCDT should utilize parallel search hardware. The remaining tables, with the exception of those in the index translation unit (IXU), are palced in DBCCP. The parallel search hardware is called the <u>security and clustering unit</u> (SCU) which has the capability of searching tables in parallel.

Let us review the role played by DBCCP. DBCCP regulates the operation of both the structure and data loops and interfaces with the front-end computer systems. It processes all DBC commands received from the front-end computer systems, schedules their execution based on the command type and priority, and routes the response set to the front-end computer systems. Additionally, it makes use of the parallel searching capabilities of SCU to search the tables needed for the enforcement of security and the clustering of records.

Architecturally and technologically, SCU is organized in a manner similar to the structure memory (SM). SCU also consists of a number of processing units. Each unit is associated with its own blocks of memory. Tables to be searched in SCU are placed in such a manner that they are spread evenly over the memory blocks of all the processing units. Thus, in searching a particular table, each processing unit needs to search only that part of the table that is in its own memory block. Since all processing units search simultaneously, the table is in effect being searched in parallel.

On the other hand, these tables could have been included in the structure memory (SM), since it also has parallel search capabilities, but this is not done for two important reasons:

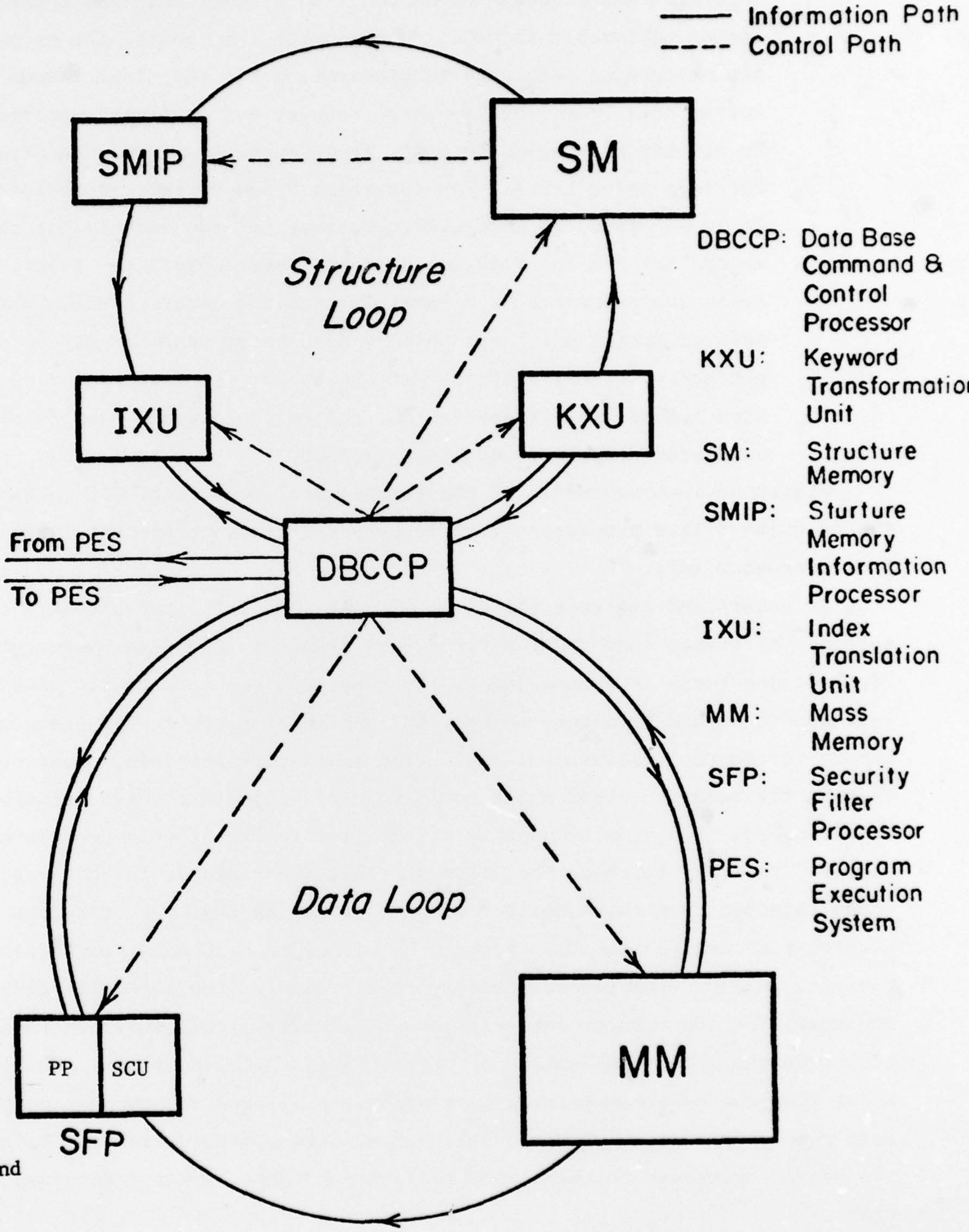
(1) It is prefereable, for the sake of improved reliability and security, to preserve the functional specialization of each component of DBC. SM is really an index to, and a condensed image of, the mass memory (MM). If it were to be used for security enforcement or data clustering, this would no longer be true.

(2) It is advantageous to assign SM only a limited set of functions in order that it does not become a bottleneck and impair the pipelining achievable in DBC. To elaborate this point, let us consider the processing necessary in preparation for the simultaneous execution of a 'retrieve-by-query' request and an 'insert-record' request. To satisfy the former request, it is necessary to search SM and retrieve index terms. For inserting a new record, on the other hand, it is necessary to determine the security atom and cluster the record belongs to by searching tables stored in SCU. Since, index terms are stored in SM separately from the security and clustering related tables which are in SCU, both these searches may be conducted concurrently. In contrast, if the tables for determining security atom numbers were stored in SM, the searches would have to be conducted serially, one after another.

Figure 43 is a schematic of the refined architecture of DBC. We note that the security filter processor (SFP) is composed of two components, SCU and the post processor (PP).

To understand the role played by PP, let us briefly recapitulate the sequence of events that occur after a user issues a 'retrieve-by-query' request. First, index terms corresponding to the query conjunctions of the query are retrieved from the structure memory (SM) and intersected in the structure memory information processor (SMIP). The index terms contain, among other things, the security atoms which could possibly contain records satisfying the user's query. The atomic access privilege list (AAPL) of this user is now accessed in order to check the allowed accesses on records in the security atoms obtained from the index terms. Let us assume that the user does not wish to read entire records. Rather, he wishes to read only two attributes, Name and Salary, from records satisfying his query. The AAPL will clearly indicate, for the records in the relevant security atoms, whether 'read' access on the above combination of attributes is allowed or not. The list of security atoms on whose records such access is allowed is now sent on to the mass memory (MM) along with the cylinder numbers obtained from the index terms, the user's query and the attribute combination [Name, Salary] required to be read.

The mass memory (MM) now retrieves the relevant records, which satisfy the security requirements and the user's query, in their entirety. These entire records are now passed on to the post processor (PP) along with the attribute combination [Name, Salary] required to be read for the user. PP



| | |
|---|---|
| DBCCP: | Data Base Command & Control Processor |
| KXU: | Keyword Transformation Unit |
| SM: | Structure Memory |
| SMIP: | Sturture Memory Information Processor |
| IXU: | Index Translation Unit |
| MM: | Mass Memory |
| SFP: | Security Filter Processor |
| PES: | Program Execution System |

| | |
|---|---|
| PP: | Post Processor |
| SCU: | Security and Clustering Processor |

FIGURE 43. The Architecture of DBC

extracts the relevant fields from the retrieved records and passes these extracted fields back to DBCCP which then passes them to the front-end computer. Before passing the extracted fields to DBCCP, PP may sort the retrieved information on some attribute. Thus, PP performs the functions of field extraction and sorting. PP may also be used to perform the (relational) join operation.

## 6.1 Certain Restrictions Placed on the User

At this point, we wish to point out certain conditions placed on a user of DBC. Suppose that a user wishes to access records which have the attributes: Name, Salary and Hometown present in their keywords. Also, let us assume that the user wishes to read only the first two of these three fields from such records. We require that the 'retrieve-by-query' request received by DBC contains a list of fields _to be read_, rather than a list of fields that _need not be read_. Thus, in the above example, the request specifies 'read Name and Salary', rather than 'read all fields except Hometown'. The reason for this should be immediately apparent to the reader. If the request is of the former kind, the AAPL may be checked a priori to see if read access on the combination [Name, Salary] is disallowed. However, if the request is of the latter kind, this a priori checking cannot be done. In this case, records satisfying the user's query must first be retrieved from MM. Then, the fields in the records must be examined. The AAPL must now be checked to see if any field or combination of fields in the records retrieved may not be read. Thus, costly a posteriori checking needs to be done and this must be avoided as it leads to access imprecision. Hence, we require the user to specify a list of fields to be read rather than a list of fields that need not be read. The only time that the user need not specify a list of fields to be read is when he wishes to read _entire_ records. In this case, it is only necessary to check, from the AAPL, if 'read' access is disallowed on _any_ attribute combination. This may be done a priori.

DBC also does not attempt to output records which may in some sense partially satisfy a query. Let us explain this by means of an example. Once again, consider that records consist of three attributes: Name, Salary and Hometown. Also, let us assume that 'read' access is denied for a particular user on this combination of these three fields. If this user requests to read entire records, this request certainly cannot be allowed. However, he may read either of the six combinations listed below.

1. [Name, Salary].
2. [Name, Hometown].
3. [Salary, Hometown].
4. [Name].
5. [Salary].
6. [Hometown].

Thus, when the user requests entire records, DBC has the choice of either denying the request, or sending to him one of the six field combinations listed above. Rather than make such an arbitrary choice, DBC will reject the request.

## 6.2 Adding and Changing Security Requirements

It is possible that, from time to time, the file creator would like to make certain changes to his file. Perhaps, the most common modification requested by a file creator is the addition of one more user to the list of users allowed to access his file. The file creator also needs to specify, for this new user, a list of field-level security descriptors, one such list for each security conjunction of the file.

Pictorially, the situations before and after the addition of a new user (user 3) are shown in Figures 44 and 45. In Figure 45, P31, ..., P3n are the lists of field-level security descriptors specified for the new user. DBC easily responds to this request by creating a new AAPL for this user and storing it in DBCCP.

A file creator may also wish to modify the security conjunctions of his file. For example, he may wish to add another security conjunction SC. This, as we know, could potentially double the number of existing security atoms. Each existing security atom could be split into two parts. One part would contain records that satisfy SC and the other part would contain records that do not satisfy SC. Such an addition on the part of the file creator would require a major operation on the part of DBC. Each record in the file must be read from MM, examined to see whether it satisfies the new security conjunction, assigned to a new security atom and reinserted into MM. All the old directory entries have to be deleted and new ones created during the reinsertion process, since the assignment of records to security atoms has now been changed. Along with the new set of security conjunctions, the file creator also needs to specify the protection requirements for each user of his file on records satisfying these security conjunctions. Thus, DBC needs to delete the old security information tables and create new ones. DBC also needs to create new AAPLs for the various users of the file.

| Security Conjunction | Field-Level Security Descriptors for User 1 | Field-Level Security Descriptors for User 2 |
|---|---|---|
| Security Conjuntion 1 | $P_{11}$ | $P_{21}$ |
| . | . | . |
| . | . | . |
| . | . | . |
| Security Conjunction n | $P_{1n}$ | $P_{2n}$ |

$P_{ij}$ is the list of field-level security descriptors for user i on records satisfying conjunction j.

FIGURE 44. A pictorial view of the access privileges allowed to the users of the file on records satisfying the various security conjunctions.

| Security Conjunction | Field-Level Security Descriptors for User 1 | Field-Level Security Descriptors for User 2 | Feild-Level Security Descriptors for User 3 |
|---|---|---|---|
| Security Conjunction 1 | $P_{11}$ | $P_{21}$ | $P_{31}$ |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |
| Security Conjunction n | $P_{1n}$ | $P_{2n}$ | $P_{3n}$ |

FIGURE 45. A pictorial view of the access privileges allowed to the users of the file on records satisfying the various security conjunction (after the addition of a new user).

DBC handles the above operations in the following manner. It first copies the file in an auxiliary store, then pretends to have received a 'delete-file' request, followed by the 'open-database-file-for-creation', 'load-attributes-information', 'load-descriptors', 'load-access-control-information' and 'close-database-file' requests, and executes the necessary algorithms. That is, DBC looks upon the process as if the user has deleted the file and created a new one. It is, of course, hoped that the file creator changes the security predicates of the file infrequently. Since security predicates partition the file into atoms, new or changed predicates require re-partition of the entire file which is both time-consuming and costly.

## 6.3 Some Miscellaneous Information in the Security and Clustering Unit (SCU)

SCU is used by DBCCP whenever records have to be loaded into MM of DBC. DBCCP does not need to use SCU in order to handle 'retrieve-by-query' requests.

We recall that SCU stores four different kinds of tables. In order to differentiate among the different kinds of tables, an additional field is created in each entry of the various tables. This field is the table identifier field, and is made to be the first field of every entry. If the entry is an entry in the SADT, then this field is made 1. If the entry is in the PCDT, this field is made 2. If the entry is in the SCDT, this field is made 3, and if the entry is in the CIDT, this field is made 4. Thus, the processors of SCU can identify the type of table by reading this first field.

Also, tables belonging to all the files known to DBC are stored in SCP. To show the correspondence between the tables and files, another field is incorporated with each entry of every table. This field (which is made to be the second field in all entries of all tables) stores the file identifier of the file for which the table has been created. Thus, for example, an entry in the SADT will now have four fields, instead of two as indicated in earlier sections.

## 7. CONCLUDING REMARKS

In the preceding sections, we have endeavored to present a complete picture of the mechanisms of security enforcement and data clustering in DBC. This is because we wish to tie up a lot of loose ends from previous technical reports and also because we present certain refinements in the architecture.

With respect to security, the most important concept is that of security atoms due to McCauley [8]. However, we have expanded the concept in order to allow for protection of records at the field (or attribute) level. All security checks are made prior to the retrieval of records, so that no access imprecision results.

DBC design exploits both existing and emerging technologies. The on-line mass memory (MM) is made from moving-head disks, perhaps the least expensive of all large-capacity on-line storage devices. The disks, however, are modified to allow parallel read-out of an entire cylinder in one revolution time, instead of one track at a time. It seems adequate that access is limited to one or a few cylinders, since single user transactions seldom refer to data beyond megabytes in size. As long as data is not physically scattered, sweeping of a large number of disk cylinders can be avoided. The physical dispersion of related data is prevented by a built-in clustering mechanism which uses information provided by the creator of the database.

The algorithms for supporting these have been discussed in great depth. The post processor (PP) will be discussed in a forthcoming report. The PP allows groups of retrieved records to be sorted on the basis of some attribute, or joined with other groups of records (a join being assumed to mean the same as a relational equality join). Furthermore, it performs field extraction for field-level protection.

## REFERENCES

[1] Banerjee, J. and Hsiao, D. K., "Performance Evaluation of a Database Computer in Supporting Relational Databases," Proceedings of the Fourth International Conference on Very Large Data Bases, Berlin, Federal Republic of Germany, September 1978, pp. 319-329; and Banerjee, J. and Hsiao, D. K., "The Use of a 'Non-Relational' Database Machine for Supporting Relational Databases," Fourth Workshop on Computer Architecture for Non-numeric Processing, Syracuse, New York, August 1978, pp. 91-98; Also available in Banerjee, J. and Hsiao, D. K., "DBC Software Requirements for Supporting Relational Databases," Technical Report OSU-CISRC-TR-77-7, The Ohio State University, Columbus, Ohio, November 1977.

[2] Banerjee, J., Hsiao, D. K., and Kerr, D. S., "DBC Software Requirements for Supporting Network Databases," Technical Report OSU-CISRC-TR-77-4, The Ohio State University, Columbus, Ohio, June 1977.

[3] Banerjee, J., Hsiao, D. K., and Ng, F. K., "Data Network - A Computer Network of General-Purpose Front-end Computers and Special-Purpose Back-end Database Machines," Proceedings of International Symposium on Computer Network Protocols, (Danthine, A., Editor), Liege, Belgium, February 1978, pp. D6-1 to D6-12; Also available in Hsiao, D. K., Kerr, D. S., and Ng, F. K., "DBC Software Requirements for Supporting Hierarchical Databases," Technical Report OSU-CISRC-TR-77-1, The Ohio State University, Columbus, Ohio, April 1977.

[4] Banerjee, J. and Hsiao, D. K., "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 3, No. 4, December 1978, pp. 347-384. Also available in Baum, R. I., Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part I: Concepts and Capabilities," Technical Report OSU-CISRC-TR-76-1, The Ohio State University, Columbus, Ohio, September 1976.

[5] Kannan, K., Hsiao, D. K. and Kerr, D. S., "A Microprogrammed Keyword Transformation Unit for a Database Computer," Proceedings of the Tenth Annual Workshop on Microprogramming, October 1977, Niagara Falls, New York, pp. 71-79; and Hsiao, D. K., Kannan, K., and Kerr, D. S., "Structure Memory Designs for a Database Computer," Proceedings of ACM 77 Conference October 1977, Seattle, Washington, pp. 343-350; Also available in Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part II; The Design of the Structure Mmeory and its Related Processors," Technical Report OSU-CISRC-TR-76-2, The Ohio State University, Columbus, Ohio, October 1976.

[6] Kannan K., "The Design of a Mass Memory for a Database Computer," Proceedings of the Fifth Annual Symposium on Computer Architecture, April 1978, Palo Alto, California, pp. 44-50; Also available in Hsiao, D. K. and Kannan, K., "The Architecture of a Database Computer -- Part III: The Design of the Mass Memory and its Related Processors," Technical Report OSU-CISRC-TR-76-3, The Ohio State University, Columbus, Ohio, December 1976.

[7] Hsiao, D. K., Kerr, D. S., Madnick, S. E., "Computer Security, Problems and Solution," to be published.

[8] McCauley, E. J. III, "A Model for Data Secure Systems, "Ph.D. Dissertation Department of Computer and Information Science, The Ohio State University, (1975). Available as Research Center Report OSU-CISRC-TR-75-2.

[9] Astrahan, M. M., et al., "System R: Relational Approach to Database Management," ACM Trans. on Database Systems, Vol. 1, No. 2, June 1976, pp. 97-137.

[10] CODASYL Data Base Task Group Report, ACM, New York, April 1971.

[11] IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, General Information Manual, GH20-1260-4.

[12] Ozkaharan, E. A., Schuster, S. A. and Smith, K. C., "RAP -- Associative Processor for Data Base Management," AFIPS Conference Proceedings, Vol. 44, 1975, pp. 379-388.

[13] Ozkaharan, E. A. and Sevcik, K. C., "Analysis of Architectural Features for Enhancing the Performance of a Database Machine," ACM Transactions on Database Systems, Vol. 2, No. 4, December 1977, pp. 297-316.

[14] Kannan, K., "The Design and Performance of a Database Computer," Ph.D. Dissertation, Department of Computer and Information Science, The Ohio State University (1977).

[15] Banerjee, J. and Hsiao, D. K., "DBC - A Database Computer for Very Large Databases," to appear in IEEE Trans. on Computers.

[16] Schkolnick, M., "Clustering Algoirthm for Hierarchical Structures", ACM Trans. on Database Systems Vol. 2, No. 1 (March 1977), 27-44.